ПРИМЕНЕНИЕ ПРАКТИК DEVOPS

Олег Скрынник

Введение

Методы управления ИТ-деятельностью не стоят на месте. Несколько десятков лет назад использовались одни подходы к разработке и эксплуатации информационных систем, сегодня — уже другие, а завтра придёт время следующих, переосмысленных способов и техник, опирающихся на новые знания, опыт и технологии. Бóльшую часть времени методы управления развиваются эволюционно, путём систематизации и оттачивания созданных ранее моделей, основанных на неких базовых принципах и постулатах. Однако, время от времени происходят скачкообразные изменения, позволяющие отдельным организациям-лидерам сделать существенный шаг вперёд в вопросах эффективности и рациональности применения информационных технологий.

На передовом крае ИТ-менеджмента находится движение DevOps (сокр. от англ. Development & Operations), названное так, в целом, довольно случайно. Новое имя собственное настолько же далеко от вкладываемого в него смысла, насколько аббревиатура ITIL далека от понятия «библиотека», а COBIT — от целей контроля (дополнительные сведения можно найти в главе «Управление ИТ-процессами и услугами»).

С публикацией COBIT 5 в 2012 году правообладатель подчеркнул, что, несмотря на то, что изначально аббревиатура COBIT являлась сокращением от «Control Objectives for Information and related Technology», теперь она является именем собственным.
Компания AXELOS, управляющая ITIL с 2013 года, также не рекомендует использовать первоначальное наименование «IT Infrastructure Library», ограничиваясь именем собственным ITIL.
Эксперты DevOps, стоявшие у истоков этого движения, признают ограниченность получившегося названия, призывая использовать более точные, на их взгляд, «BizDevOps», «DevSecOps» и подобные. Однако, вероятность изменения названия в настоящее время является незначительной.

Можно утверждать, что появлению DevOps в наибольшей степени способствовали два фактора: развитие гибких методов разработки программного обеспечения и управление ИТ-инфраструктурой, как программным кодом. Рассмотрим каждый из них.

Развитие гибких методов разработки программного обеспечения

В конце прошлого столетия доминирующей методологией разработки программного обеспечения была так называемая «водопадная модель» (или «каскадная модель»): последовательное выполнение заранее определённых этапов, каждый из которых занимает существенное время и завершается достижением заранее согласованных результатов, при этом переход на следующий

этап во многих случаях происходит после полного и формального завершения предыдущего этапа (Рис. 3.5.1.).

Рис. 3.5.1. Пример водопадной модели разработки ПО.

Дополнительным отличительным признаком такой модели является функциональная специализация исполнителей отдельных этапов: аналитиков, архитекторов, разработчиков, тестировщиков и т.д.

При разработке крупных информационных систем с конечной функциональностью, которую возможно определить и зафиксировать в самом начале работ, а также при отсутствии требования максимально быстрого завершения полного цикла разработки такая модель позволяет получать качественные выходные результаты при достаточно детальном контроле расходов.

Однако, в конце 90-х годов XX века, с бурным развитием Интернет-технологий и web-программирования, недостатки водопадной модели стали негативно влиять на взаимодействие и взаимопонимание заказчиков (бизнес-подразделений компании, либо внешних организаций) и исполнителей (программистов компании, либо внешних разработчиков программного обеспечения). Действительно, появляющиеся рыночные возможности для основного бизнеса требовали быстрого — за считанные месяцы — вывода на рынок новых продуктов. В то время как типичный цикл разработки от начала проекта до получения первого работающего

результата занимал от 6 до 18 месяцев, а в крупных организациях — до 2-3 лет. Кроме того, в условиях появления ранее неизвестных, но потенциально перспективных рыночных возможностей требования заказчиков могли меняться по ходу проекта разработки, что было крайне сложно учесть при создании ИТ-системы без увеличения сроков, либо снижения качества выходных результатов (Рис. 3.5.2.).

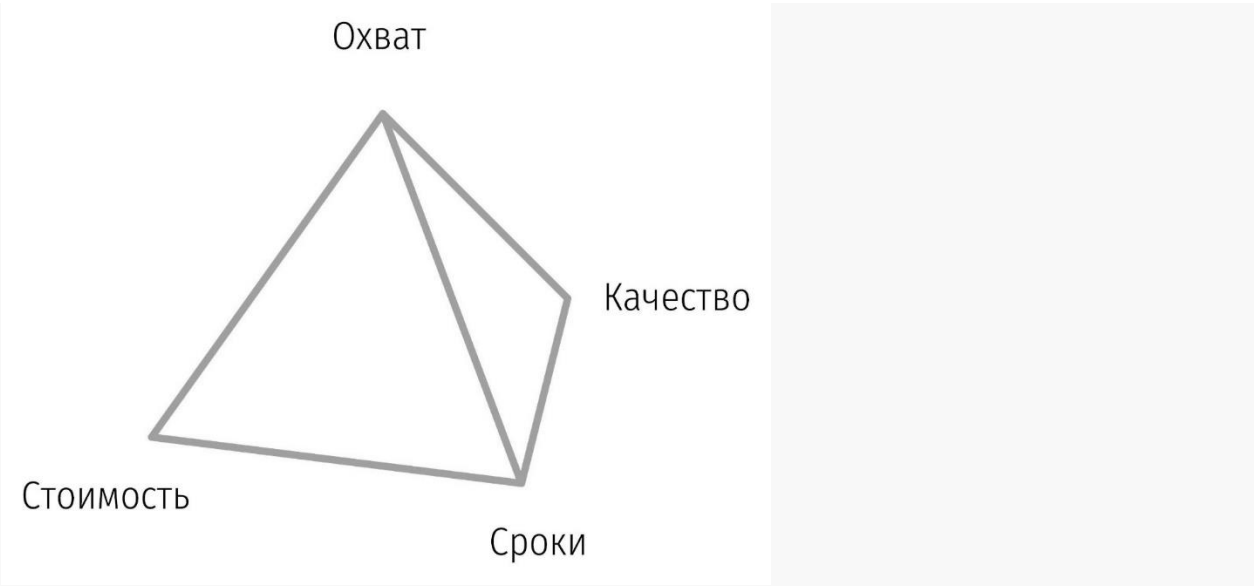

Рис. 3.5.2. Классическая пирамида взаимосвязанных ограничений проектного управления

Таким образом, накапливалось напряжение между заказчиками и исполнителями, между основным бизнесом и разработчиками ПО. Ответом на такой вызов стали инновационные подходы к программированию. К. Швабер выпустил несколько публикаций о Scrum (например, «Agile Software Development with Scrum», K. Schwaber, 2001, ISBN: 978-0130676344). К. Бек опубликовал книгу об экстремальном программировании — XP («Extreme Programming Explained: Embrace Change», K. Beck, 1999, ASIN: B01FKT01PY). Однако, применение новых идей давало весьма скромные результаты, в основном потому, что такое применение фокусировалось лишь на одном из этапов цикла разработки ПО — на, собственно, программировании, при том, что задача ставилась намного более широкая. Требовалось что-то, что позволит упростить и ускорить весь жизненный цикл ПО.

В 2001 году К. Швабер, К. Бек, а также ещё пятнадцать экспертов встретились, чтобы обсудить имевшиеся проблемы и выработать решение. Итогом стал так называемый манифест Agile, призванный устранить разрыв понимания между бизнесом и разработчиками ПО (дополнительные сведения можно найти в главе «Управление разработкой ПО»).

Последовавшее развитие и принятие идей гибкой разработки сообществом программистов и менеджеров проектов сильно ускорили и перестроили разработку ПО.

Ключевыми элементами гибкой разработки являются более плотное взаимодействие между заказчиком и исполнителем, уменьшение размера задач, ритмичность выдачи результатов через короткие промежутки времени (циклы) и ограничение размера команд.

Группа разработки ПО, применяющая гибкие подходы, выдаёт готовый к эксплуатации новый код каждые две-четыре недели. Конечные потребители плотнее вовлечены в создание продукта, а, значит, быстрая обратная связь значительно влияет на дальнейшее развитие продукта, что дополнительно добавляет вкуса к быстрым изменениям.

Однако, во многих компаниях отказ от водопадной модели в пользу гибкой разработки даёт куда меньший эффект, чем ожидается. Такие наблюдаемые в работе многих компаний результаты связаны не столько с какими-то преимуществами водопадной модели или недостатками Agile. Зачастую, полезный эффект нивелируется тем, что разработка кода — лишь одно из звеньев в цепочке создания ценности.

Действительно, до начала самой разработки имеется ещё значительный блок работ, направленный на выявление бизнес-потребностей, их проработку, анализ, приоритизацию и т.д.

Далее, по окончании разработки, готовый программный код необходимо быстро развернуть в среде эксплуатации, чтобы заказчики получили всю ту пользу, которую им обещали, а также могли предоставить обратную связь разработчикам относительно качества получившегося продукта. При этом, почти во всех организациях, возникших до 2010-х годов, ИТ-инфраструктура является жёсткой, основанной на дорогом аппаратном обеспечении, которое было приобретено достаточно давно, бюджеты на закупку и настройку выделялись непросто, да и бюджетный процесс для новых закупок — долгий.

Более того, в подавляющем числе организаций ИТ-инфраструктура находится в довольно хрупком состоянии. Одним из факторов, усиливающих такую хрупкость, является комплексность, сложность применяемых ИТ-решений. Используется множество, десятки тысяч взаимосвязанных компонентов. Другим фактором служит слабое документирование, равно как и быстрое устаревание документации относительно применяемых ИТ-решений и ИТ-систем, в том числе устаревание знаний ИТ-персонала, а также потеря знаний вследствие текучки кадров.

Трогать ИТ-инфраструктуру во многих компаниях страшно. Изменение — самое большое зло для отдела эксплуатации ИТ-систем, а постоянный большой поток изменений может привести к катастрофическим последствиям.

Таким образом, передовые методы разработки ПО упираются в барьеры со стороны подразделений, ответственных за эксплуатацию информационных

технологий, что нивелирует возможный положительный эффект применения гибких подходов.

Управление ИТ-инфраструктурой как программным кодом

Возникновению управления ИТ-инфраструктурой как программным кодом предшествовало появление и развитие двух технологий: виртуализации и облачных вычислений.

История виртуализации программных и аппаратных сред началась довольно давно — в 1964 году, с началом разработки операционной системы IBM CP-40. За годы последовательного развития этого направления был достигнут весьма значительный прогресс. Коммерчески доступные системы появились для мейнфреймов (70-80-е годы прошлого века) и для более распространённых в последующем машин на архитектуре Intel x86 (80-90-е годы).

Виртуализация позволила не только более эффективно использовать дорогое и мощное аппаратное обеспечение, но и ввести дополнительный уровень абстракции между исполняемым кодом, предоставляющим полезные результаты заказчику, и нижележащим системным программным обеспечением. Был сделан существенный шаг в сторону разделения компетенции и ответственности между, условно говоря, «прикладниками» и «системщиками», в широком смысле данных понятий.

Технология облачных вычислений развивалась ещё быстрее. До середины 90-х годов прошлого века телекоммуникационные компании предлагали своим клиентам организацию частных глобальных вычислительных сетей (WAN — Wide Area Network) путём прокладывания соответствующих соединительных кабелей для каждой точки, каждого заказчика, от пункта А до пункта Б. Но с появлением технологии частных виртуальных сетей (VPN — Virtual Private Network) возникла возможность по одним и тем же каналам передачи данных отправлять пакеты разных клиентов, обеспечивая должный уровень безопасности, приватности и качества сервиса. Именно тогда для наглядного отображения разграничения ответственности — где идёт «кабель от клиента», а где трафик попадает в общую разделяемую сеть, — провайдеры стали использовать символ облака.

Клиенты, получившие возможность передачи данных на большие расстояния, стали использовать данные технологии не только для собственно обмена информацией между своими территориально удалёнными друг от друга системами, но и для распределения вычислительной нагрузки между разными узлами своих сетей. Напрашивалось появление технологии, упрощающей и удешевляющей такое взаимодействие. Небольшие провайдеры сделали первые шаги, а действительно масштабные изменения случились в 2006 году, когда компания Amazon представила своё решение ECC (Elastic Compute Cloud). Вскоре, в 2008 году, компания Microsoft запустила свой сервис Azure, а компания Google — сервис

Google App Engine, впоследствии развившийся в Google Cloud Platform. Это, разумеется, не единственные, но самые крупные примеры предоставления вычислительных мощностей в аренду.

Виртуализация и облачные технологии сильно изменили вычислительный ландшафт. Предлагаемые коммерческими провайдерами ресурсы стали доступными по стоимости, надёжными и обеспечивающими необходимый уровень безопасности. Отношение клиентов к облакам и их использованию изменилось от «кто-то другой где-то управляет моим железом» на «у меня есть инфраструктура, которой я управляю на расстоянии» (дополнительные сведения можно найти в главе «Управление ИТ-инфраструктурой»).

Что же это означает — управлять инфраструктурой на расстоянии? Вспомним одну из ключевых парадигм Unix-систем: все необходимые действия с системой можно произвести из командной строки (а значит — и с помощью скрипта). Графические оболочки являются красивым, но опциональным инструментом.

Объединим теперь виртуальные облачные технологии и интерфейс командной строки для всех задач. В результате, ИТ-специалисты получили возможность с помощью текстовых команд создавать необходимые части ИТ-инфраструктуры, включая серверы, системы хранения данных, сетевые компоненты, все интерфейсы между ними, все настройки и конфигурации… Степень автоматизации существенно возросла, равно как и скорость выполнения необходимых изменений. Раньше для разворачивания ИТ-инфраструктуры, основанной на собственном аппаратном обеспечении, требовалось:

- обосновать и согласовать бюджет (недели и месяцы);
- дождаться очередного цикла закупки (месяцы);
- заказать оборудование у поставщика и оплатить его (дни);
- дождаться поставки (недели и месяцы);
- получить, установить, настроить, подготовить к использованию (дни и недели).

Теперь аналогичную по характеристикам ИТ-инфраструктуру можно создать так:

- запустить скрипт, дождаться окончания его выполнения (минуты, редко — часы);
- оплатить счёт облачного провайдера в конце месяца.

То есть, необходимая инфраструктура создаётся с помощью программного кода. И не только создаётся, но и может управляться как программный код — с хранением версий, отслеживанием изменений, отладкой, повторным использованием прошлых наработок и т.д.

В завершение отметим также вторую жизнь, которую получили давно придуманные технологии. К примеру, виртуализация на уровне операционной системы была доступна во многих UNIX-системах ещё в 80-е годы прошлого столетия. Однако,

серьёзный коммерческий успех этой технологии, которую чаще стали называть контейнеризацией, пришёл только во второй половине 2000-х, что совпадает по времени с событиями, описанными ранее. И если изначальный механизм chroot был довольно ограничен по функциональности и возможностям, то сейчас для контейнеров можно изолировать файловую систему, выделять дисковые квоты, ограничивать предоставляемые оперативную память, время процессора, ширину каналов ввода-вывода и т.д.

Неизбежность появления

Рассмотренные истоки возникновения DevOps позволяют сделать следующие выводы.

Во-первых, из-за появления новых способов взаимодействия с основным бизнесом, клиентами, и грамотного применения методов гибкой разработки назрела потребность строить работу и управление информационными технологиями иначе.

Во-вторых, с возникновением новых технологий управления инфраструктурой появилась возможность строить работу ИТ иначе.

Можно предполагать, что появление чего-то, аналогичного DevOps, было лишь вопросом времени.

Задачи, решаемые с помощью DevOps

Методология DevOps призвана решить три вполне конкретные задачи современной ИТ-организации.

Ускорение вывода на рынок

Компании, применяющие DevOps, наиболее часто сообщают о необходимости существенно сокращать время вывода на рынок (англ. Time to market). Под этим термином разные люди подразумевают разное. Часто встречающееся понимание — время от зарождения какой-либо бизнес-идеи до возможности клиенту приобрести новый продукт или получить новую услугу, являющуюся результатом воплощения бизнес-идеи в жизнь. Таким образом, в расчёт (а точнее — в оценку) времени вывода на рынок включается довольно большой промежуток, содержащий в случае необходимости привлечения ИТ-департамента следующие шаги:

- структурирование и первое формальное описание бизнес-идеи, а скорее — нескольких бизнес-идей, их обоснование;
- оценка и выбор бизнес-идеи для реализации;
- планирование необходимых действий для реализации, выделение финансирования;
- подготовка бизнес-процессов и персонала;
- одновременно с этим: формализация требований, разработка прототипа, первичное тестирование, разработка полнофункциональной ИТ-системы, её тщательное тестирование, передача в эксплуатацию, запуск, тиражирование;
- одновременно с этим: маркетинговые активности, подготовка рынка, подготовка механизма и каналов продаж;
- запуск нового продукта или новой услуги.

Описанному процессу присущи некоторые сложности.

Во-первых, его общая длительность может составлять годы, при том, что бизнесу хотелось бы сократить её до месяцев. Бизнес-обоснование здесь прозрачно: за время разработки нового продукта рынок может измениться настолько, что продукт будет уже неактуален, либо конкуренты выпустят аналогичный продукт раньше, соберут сливки и закрепятся как лидеры. Ранний выход на рынок с привлекательным конкурентным предложением помогает занять доминирующее положение в новых нишах, которое, в свою очередь, даёт лидеру возможности в дальнейшем изменять рынок, подстраивая его под себя. Это существенное преимущество, которым обладают немногие, хотя стремятся к нему все. Кроме того, не следует забывать про всё возрастающую скорость изменений. Одна из наиболее наглядных иллюстраций данного тезиса — закон ускоряющихся возвратов (Law of Accelerating Returns), сформулированный в 1999 году Р. Курцвайлом. Согласно ему, скорость изменений в широком спектре эволюционирующих систем, включая новые технологии, но не ограничиваясь ими,

стремится расти экспоненциально. На практике это означает, что прорывы в технологиях, в том числе информационных, случаются всё чаще. Компании, которые увеличивают темп изменений, становятся лидерами, а те, кто лишь могут сохранять свой быстрый темп, получают возможность не остаться на обочине. Что уж говорить про тех, кто не способен меняться быстро…

Вторая сложность описанного выше процесса заключается в необходимости чёткой координации и согласования взаимозависимых шагов, особенно выполняющихся параллельно. В этот момент многие компании попадают в классическую ловушку: пока нет готового продукта — нечего рекламировать и продавать, однако с появлением такового начало маркетинговых активностей приводит к продажам (а значит — и к финансовой отдаче) лишь с задержкой. Такая ловушка ещё больше увеличивает фактическое время вывода на рынок и требует ещё более аккуратной координации всех исполнителей.

Отметим, что роль традиционного ИТ-подразделения в увеличении срока вывода на рынок трудно переоценить. Действительно, в некоторых организациях из общего календарного времени в 1,5-2 года на ИТ-работы приходится более 50-70%.

Другое понимание термина «время вывода на рынок» менее глобально, но не менее значимо. Динамичные компании, создающие цифровые продукты, привыкли действовать быстро. Скрупулёзному и детальному планированию они предпочитают эксперименты, а слово «идея» заменяют на «гипотезу». В этом случае процесс выглядит примерно так:

- рождение гипотезы, разработка методов оценки её справедливости;
- реализация гипотезы на практике;
- измерение результата, A/B тестирование, сравнение с целевыми значениями;
- корректировка по итогам анализа, переход на первый или второй шаг.

Несложно заметить возникновение цикла, ожидаемая скорость которого — недели. Такой быстрый темп необходим потому, что сама суть движения — в постоянном поиске. На старте, в самом начале, совершенно неизвестно конечное состояние, и, тем более, неизвестна дорога к нему. Долгосрочное планирование не имеет никакого смысла, компания видит лишь следующий, ближайший шаг — точнее, пытается его угадать. Проиллюстрировать данный тезис поможет широко известная метафора, сравнивающая выживание и развитие бизнеса с поиском реки с деньгами (Рис. 3.5.3).

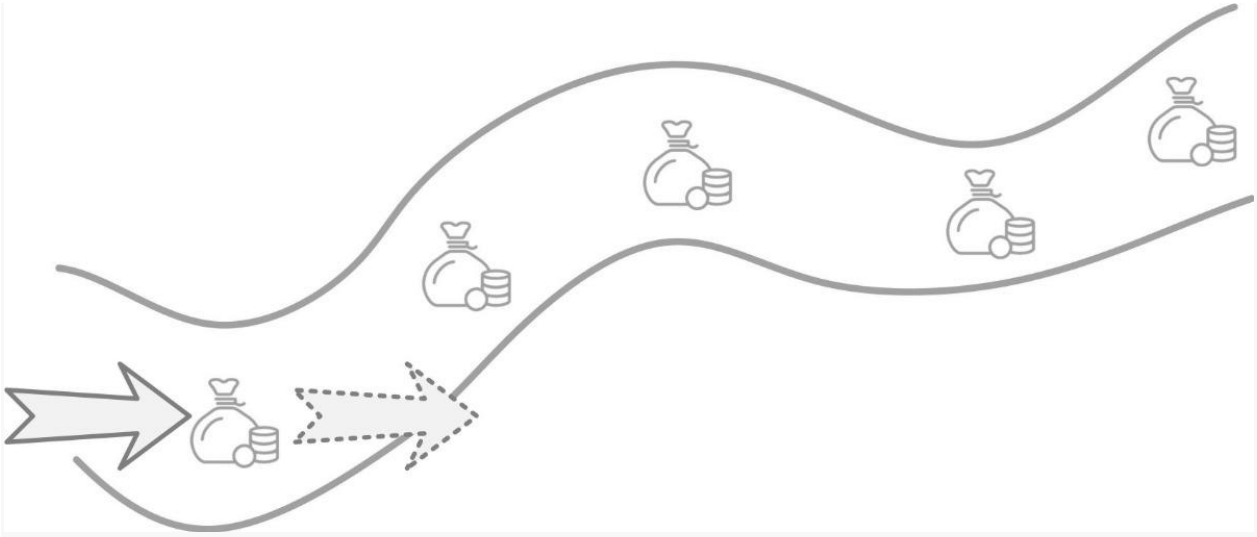

Рис. 3.5.3. Река с деньгами.

Один раз войдя в такую реку, нащупав новую нишу и новые возможности, компании будет необходимо всегда искать изменяющееся русло. При том, что традиционные процессы, регламенты, уже имеющиеся продукты будут с большой вероятностью увеличивать инерцию компании и, оставленные без внимания, приведут к выходу на берег.

Нетрудно догадаться, что вклад ИТ-департамента в замедление приведённого выше цикла высок. Действительно, в создании цифровых продуктов роль ИТ — ключевая, поэтому задержки на этапе реализации гипотезы в наибольшей степени происходят именно благодаря «медленному» ИТ-отделу, предлагающему вместо ожидаемых недель — месяцы.

Для уменьшения времени вывода на рынок DevOps предлагает множество техник, например: уменьшение размера задач, уменьшение количества передач работы, постоянные поиск и устранение потерь и др. Однако, важно сделать следующее замечание: наивно надеяться, что применение техник DevOps для ускорения работы ИТ-отдела одновременно приведёт к сокращению затрат на ИТ. Скорее, наоборот — расходы на информационные технологии вырастут, что обусловлено, в первую очередь, увеличением численности ИТ-персонала. Действительно, традиционная организация ИТ-отдела предполагает наличие отдельных функциональных подразделений, каждое из которых занимается всеми задачами в рамках своей предметной области (бизнес-анализ, разработка и тестирование, эксплуатация, поддержка, развитие и т.д.). При этом, внутри каждого такого функционального подразделения обеспечивается необходимая взаимозаменяемость специалистов, а среднее и большое число специалистов одинаковых квалификации и компетенций позволяют равномерно распределять между ними нагрузку (Рис. 3.5.4).

Рис. 3.5.4. Функциональная структура традиционного ИТ-подразделения.

В отличие от такой схемы, в DevOps деление специалистов производится по командам, и каждая команда работает над своим продуктом. Будучи самодостаточной, команда включает в себя и владельца продукта, и архитекторов, и разработчиков, и тестировщиков, и ответственных за эксплуатацию, и за информационную безопасность (Рис. 3.5.5.).

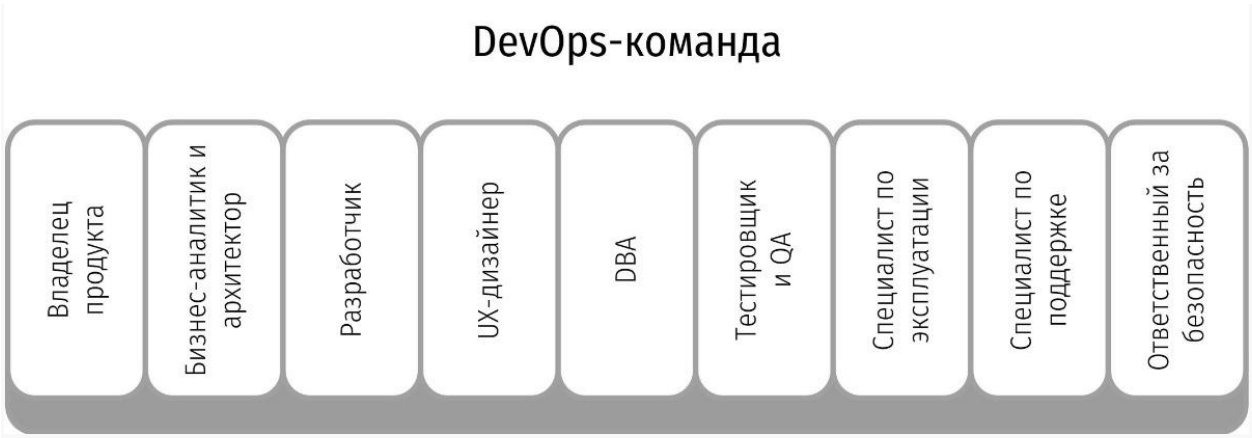

Рис. 3.5.5. Пример состава DevOps-команды.

При большом количестве команд, каждая из которых сфокусирована исключительно на своём продукте, равномерность загрузки специалистов обеспечить сложнее, что может приводить к неполной утилизации персонала, а значит — к повышению расходов на него.

Таким образом, можно утверждать, что традиционная организация ИТ-подразделения больше направлена на оптимизацию затрат (англ. Optimize for cost), в то время как организация по DevOps направлена на оптимизацию скорости (англ. Optimize for speed), и данные цели в общем случае являются разнонаправленными.

Отметим также, что DevOps предлагает инструменты и способы ограничения роста затрат, такие как максимальная автоматизация всех рутинных операций, а также взаимозаменяемость в пределах одной команды. Кроме того, адепты DevOps справедливо указывают, что оптимизация скорости во многих случаях направлена на предоставление возможности бизнесу зарабатывать больше, что компенсирует возрастающие расходы на ИТ. В таком случае можно рассуждать об ИТ-отделе, как истинном бизнес-партнёре, а не центре затрат.

Снижение технического долга

Понятие технического долга предложил У. Каннингем в 1992 году . Возникновение такого долга происходит, когда программист выбирает неоптимальный путь решения задачи для того, чтобы сократить сроки разработки. Уорд отмечал, что это естественный процесс, и, собственно, проблема заключается в том, что накапливающиеся неоптимальные решения приводят к постепенному ухудшению результатов разработки, и, как следствие, к деградации продукта. Со временем команда разработки будет вынуждена больше времени уделять исправлению последствий ранее принятых решений, то есть переделке кода, нежели разработке новых функциональных возможностей. Аналогия с финансовым долгом в этом случае является очень наглядной — для ускорения получения результата компания может «влезть в долги», однако, она не должна допускать ситуации, когда вся получаемая прибыль уходит на обслуживание долга.

Мартин Фаулер (Martin Fowler) в дальнейшем развил идею технического долга, предложив условную классификацию причин его возникновения (Табл. 3.5.1).

Табл. 3.5.1. Классификация технического долга по М. Фаулеру (источник https://martinfowler.com/bliki/TechnicalDebtQuadrant).

	Беспечно	**Рассудительно**
Преднамеренно	*«У нас нет времени на разработку архитектуры»*	*«Мы должны выпустить продукт как можно скорее, осознавая последствия»*
Нечаянно	*«Что такое инкапсуляция?»*	*«Теперь мы знаем, как нам стоило это сделать»*

Его точка зрения в целом повторяет мысль У. Каннингема — в правильно организованной команде разработчиков увеличение технического долга может быть осознанным шагом для получения краткосрочных преимуществ; важно уделять внимание «выплате» этого долга.

В настоящее время понятие технического долга обычно употребляется намного более широко. При расширении его применения на вопросы эксплуатации поднимается целый пласт проблем традиционного ИТ-отдела: устранение сбоев с помощью перезагрузки устройств; установка программной заплатки, не

протестированной должным образом; выполнение изменений ИТ-инфраструктуры без тщательного планирования; ручное исправление какого-либо скрипта или настройки сервера без документирования — это лишь отдельные примеры накопления технического долга, который в обычном ИТ-отделе никто никогда не будет «выплачивать». Некоторые ИТ-организации даже не планируют таких работ или проектов, другие тешат себя иллюзиями наведения порядка, как только для этого появится свободная минута — разумеется, свободной минуты в современном ИТ-подразделении не появляется.

Более того, можно утверждать, что некоторые общеизвестные практики, предлагаемые библиотекой ITIL, будучи применёнными неграмотно или изолированно, могут также приводить к увеличению технического долга. Например, процесс управления инцидентами, согласно ITIL, не имеет цели поиска и устранения причин возникновения сбоев. Его задача — скорейшее восстановление работы ИТ-системы (или ИТ-услуги, не принципиально), в т.ч. с помощью применения обходных, временных решений. Применение таких решений практически гарантирует повторение сбоев, а значит — новые затраты ИТ-организации на повторное их устранение. Авторы ITIL предполагали, что параллельно процессу управления инцидентами в организации будет работать процесс управления проблемами, чья задача — поиск и устранение корневых причин возникновения инцидентов: по сути, снижения технического долга в его широком понимании. Однако заметим, что в большинстве современных ИТ-отделов есть хоть как-то работающий процесс управления инцидентами, в то время как увидеть в дикой природе процесс управления проблемами наоборот, крайне сложно.

DevOps уделяет пристальное внимание вопросам снижения технического долга, а точнее — управлению им. Для примера можно привести две часто применяемые практики. Во-первых, постоянно выполняемый рефакторинг программного кода позволяет учитывать полученный при эксплуатации опыт, а работы по устранению ранее допущенных (осознанно или случайно) узких мест планируются наравне с созданием новой функциональности. Во-вторых, DevOps настоятельно рекомендует применять практику «проблемные шаги повторять как можно чаще», чтобы не допускать «застаивания» проблем, о которых все знают, но ни у кого не доходят руки их устранить.

Устранение хрупкости

Как уже упоминалось ранее, ИТ-инфраструктура большинства организаций находится в весьма шатком состоянии. Это обусловлено многими причинами, действующими в совокупности:

- технические решения создавались постепенно, годами, из разных составляющих;
- применяются большие системы сторонней разработки, сильно кастомизированные под задачи данной компании;

- применяются системы собственной разработки, при том, что и ключевые программисты, и команды целиком уже могут в компании не работать;
- настроено большое количество разнообразных интеграций систем между собой, а также с внешними источниками и потребителями данных;
- применяемые решения, не всегда оптимальны ввиду необходимости ускорения их реализации, а также ограничений бюджета;
- текущие работы по эксплуатации и поддержке добавляют временных, обходных решений, «костылей», только чтобы всё это продолжало работать дальше;
- документирование программного кода, архитектуры, инфраструктуры, технических решений и даже контрактных обязательств оставляет желать лучшего.

Дж. Ким, Дж. Хамбл, П. Дебуа и Дж. Виллис отмечают («The Devops Handbook: How To Create Worldclass Agility Reliabil-ity And Security In Technology Organizations», G. Kim, J. Humble, P. Debois, J. Willis, 2016, ISBN 978-1-942-78800-3, раздел «Downward Spiral In Three Acts»), что, по злой иронии, наиболее хрупкими являются именно те системы и приложения, от которых бизнес зависит в наибольшей степени, и которые приносят ему максимальную пользу. Уменьшать хрупкость таких систем крайне сложно ввиду больших рисков нарушения работы бизнеса, недопустимости простоя, а также постоянного потока новых изменений и доработок, связанных именно с этими системами.

Но и продолжать работать с такой неустойчивой инфраструктурой опасно для карьеры ИТ-руководителей и ИТ-менеджеров. Кроме того, помимо долгосрочных нависающих неприятностей, есть и оперативные сложности — внесение любых изменений является риском, а потому необходимы соответствующие инструменты его снижения: долгое и тщательное обоснование необходимости, планирование, согласование и утверждение, проработка, тестирование и, наконец, выполнение. Всё это существенно замедляет выполнение изменений, а также негативно отражается на способности ИТ-организации к инновациям.

DevOps предлагает бороться с хрупкостью ИТ-систем самым радикальным образом — путём её тотального устранения. В традиционной парадигме новый программный код находится в нерабочем состоянии до тех пор, пока тестирование не докажет его работоспособность. В DevOps, напротив, и код, и система в целом в любой момент времени полностью работоспособны, и, если очередное изменение нарушает такую работоспособность, оно немедленно откатывается назад — система же продолжает работать исправно.

В своей книге «Антихрупкость: как извлечь выгоду из хаоса» («Antifragile: Things That Gain from Disorder», N. Taleb, 2012, ISBN 978-1400067824) Н. Талеб рассуждает об особенностях сложных систем и вводит следующую классификацию: хрупкие системы, устойчивые системы и антихрупкие системы. Приведённое разделение помогает выбрать принципиальный подход к работе: хрупким системам в первую очередь нужна стабильность, их нужно как можно реже менять, а изменения

тщательно проверять как до, так и после вмешательства. Устойчивые системы проектируются с учётом присущей им сложности и хрупкости, в них закладываются механизмы отказоустойчивости и выживания, позволяющие в процессе эксплуатации и при изменениях меньше беспокоиться о возможных негативных последствиях. Но наиболее совершенны так называемые антихрупкие системы, улучшающиеся при столкновении со сбоями и беспорядком (то есть — с реальностью корпоративных информационных технологий).

Одна из замечательных практик DevOps, связанная с антихрупкостью — намеренное внесение хаоса и нестабильности в среду эксплуатации. Такая техника известна под разными названиями: игровой день (англ. Game Day), обезьяна хаоса (англ. Chaos Monkey), армия обезьян (англ. Simian Army), но суть сохраняется без изменений. Специально разработанные программные средства нарушают работу ИТ-систем, серверов, систем передачи и хранения данных и т.д. — случайным образом в неизвестные заранее моменты времени. Целевые ИТ-системы должны в ответ самостоятельно и максимально оперативно обнаруживать неисправность и восстанавливать свою работоспособность, в идеале таким образом, чтобы конечный пользователь ничего не заметил, а данные, разумеется, не были утеряны. Такую технику можно попробовать использовать и в традиционном ИТ-отделе, однако, во многих компаниях она может привести к полному блокированию работы бизнеса.

Итак, рассмотрены три основные задачи, которые ставятся перед DevOps: уменьшение времени вывода на рынок, снижение технического долга и устранение хрупкости. Решение каждой из них по отдельности способно дать существенные преимущества современному бизнесу, но три вместе представляют собой мощный драйвер изменений. Рассмотрение каждой из задач завершалось коротким упоминанием практик DevOps, помогающих в достижении соответствующих целей. Заметим теперь, что само по себе применение указанных практик не приведёт к решению обозначенных задач, этого недостаточно. Необходимо самым серьёзным образом менять культуру работы ИТ-организации, чтобы изменились не только применяемые инструменты, приёмы и техники, но и отношение ИТ-персонала к ключевым вопросам работы компании: роли заказчика, ценности информационных технологий, толерантности к известным недостаткам, необходимости постоянного совершенствования. Слепое применение идей DevOps — например, «давайте построим конвейер, ведь без него DevOps не бывает» — с большой вероятностью приведёт к явлению, известному как **культ карго** (англ. Cargo cult).

Некоторые частые заблуждения

Важно рассмотреть некоторые, наиболее часто встречающиеся, заблуждения относительно понятия DevOps. Это поможет яснее очертить границы явления и позволит перейти к рассмотрению следующих, более специфичных вопросов. Не имея задачи по наиболее полному охвату всех встречающихся недопониманий, для данного раздела были отобраны именно те из них, которые помогают понять, **что такое DevOps** с управленческой точки зрения, путём сравнения с тем, **чем DevOps не является**.

DevOps — это часть Agile

Любители современных подходов к разработке программного обеспечения иногда заявляют, что DevOps — не более, чем продолжение идей Agile. В основе такого ограниченной картины мира лежит тот факт, что гибкая разработка позволяет отлично выстроить отношения с бизнесом в части понимания его требований к программному продукту, а также достаточно быстро выдать такой программный продукт. Давняя проблема «Что с готовым продуктом делать, чтобы он приносил пользу, и как его, собственно, эксплуатировать» теперь имеет решение: у нас есть DevOps! Там и будут кем-то найдены ответы на эти неудобные вопросы.

Это, безусловно, очень ограниченный взгляд на DevOps, минимум по трём причинам. Во-первых, основываясь в существенной мере на Agile, DevOps, тем не менее, расширяет идеи гибкой разработки до гибкого ИТ-производства в целом — на всю организацию, на весь процесс, на всю цепочку создания ценности. Во-вторых, получение отдачи от DevOps требует более значительных культурных изменений в компании, чем это обычно происходит при применении Agile. В-третьих, задачи, которые ставятся перед DevOps, не ограничиваются лишь ускорением поставки — есть также необходимость снижения технического долга и устранения хрупкости.

DevOps — это автоматизация и инструменты

Другая точка зрения сводится к слову «автоматизация». Программных инструментов, помогающих работать современному ИТ-отделу, в последние годы развелось великое множество — они исчисляются сотнями. Многие вендоры будет уверять вас, что именно они и есть DevOps, либо что их инструменты тот самый DevOps обеспечат.

Маркетинговое давление вендоров очень велико. К ним уже присоединились большие компании, с большими целями по выручке и соизмеримыми бюджетами на рекламу. Многие могут заметить прямую аналогию с историей двадцатилетней давности с программным обеспечением по управлению ИТ-услугами — тогда производители ПО тоже вовсю заявляли, что ITSM — это программное

обеспечение, надо только его инсталлировать, и процессы появятся сами собой. Лишь немногие видят и всерьёз обсуждают что-то за пределами ПО.

DevOps, действительно, зависит от наличия и работоспособности определённых инструментов автоматизации. Но, строго говоря, минимальный набор таких инструментов сводится к системе контроля версий для хранения всех исходных кодов и данных о конфигурации ИТ-инфраструктуры, плюс к системе автоматизации конвейера поставки ПО. Всё остальное, как принято говорить, можно добавить по вкусу. В то время как отдельные программные пакеты широко распространены, универсального списка программных инструментов DevOps, обязательных к применению, нет и быть не может.

DevOps — это новая профессия

Следующий вариант подсказывают нам кадровые агентства и сайты размещения объявлений о работе. DevOps — говорят они — это универсальный солдат, способный и код писать, и тесты создавать, и среды разворачивать, и с ИТ-инфраструктурой управляться. То есть, он может эффективно выполнять работу и программиста, и поддержки, получая при этом только одну зарплату.

Другой часто встречающийся случай — это подмена известной древней профессии «системный администратор» на более модную «DevOps-инженер». В таких вакансиях уже из описания становится понятно, что речь вовсе не о DevOps.

Третий случай — DevOps-гуру, который необходим для «внедрения» этого DevOps в конкретной компании. Примерно, как Agile-коуч или Scrum-мастер.

Всё это, разумеется, серьёзные заблуждения. DevOps — глубокое изменение основ работы ИТ-подразделения, которое невозможно выполнить, наняв некоторое количество DevOps-инженеров или пригласив DevOps-гуру. Умение построить технический конвейер поставки ПО не гарантирует успеха. Сэкономить финансовые ресурсы, применяя практики DevOps, скорее всего не получится, как было показано ранее.

Принципы DevOps

Словом «**принципы**» обозначим ключевые идеи, на которых базируется весь DevOps, без принятия и применения которых от DevOps остаётся совсем мало смысла.

Под «**практиками**» будем понимать действия, выполняемые в соответствии с принципами, направленные на получение полезного эффекта.

Принципы будут неизменны для любой организации, применяющей идеи DevOps, в то время как практики, скорее всего, будут выбраны и видоизменены в зависимости от конкретной ситуации, компании и решаемых задач.

Все основные принципы, описываемые международными экспертами DevOps, приведены далее.

Поток создания ценности

Одно из ключевых понятий DevOps, заимствованное из бережливого производства — поток создания ценности (англ. Value Stream). Это понятие используется довольно давно, однако, по мере расширения его применения к решению практических задач, появляются новые издания, достаточно полно рассматривающие поток с прикладной точки зрения.

Считается полезным рассматривать работу организации с точки зрения создания ценности в ответ на запрос потребителя. Действия, выполняемые для реализации запроса, выстраиваются в последовательность, называемую потоком создания ценности. Обычно, в организации обрабатывается множество различных запросов. В то же время, традиционная организация работает над несколькими продуктами или услугами. Таким образом, и потоков создания ценности в компании много.

Английские слова «**Stream**» и «**Flow**» зачастую переводятся на русский язык одинаково как «поток», в то время как существует, пусть сложно различимая, но всё же разница между этими понятиями. Для её подчёркивания в настоящем издании «Flow» переводится как «течение».

Для знакомства с Value Stream можно рекомендовать следующие издания:

M. Rother, J. Shook, «Learning to See: Value-Stream Mapping to Create Value and Eliminate Muda», Lean Enterprise Institute, 2009, ISBN 978-0966784305.

K. Martin, M. Osterling, «Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation», McGraw-Hill, 2014, ISBN 978-0071828918.

Работа по моделированию потока называется **картированием** (англ. Value Stream Mapping). Она начинается с выбора одного из продуктов: иногда с того, где руководству видятся наибольшие возможности по оптимизации, а иногда с того, где коллективу представляется возможным быстро добиться существенных улучшений, заодно изучив данную технику. Построение выполняется в два шага:

сначала создаётся картина «как есть», затем — «как будет». Проработка будущего состояния важна по двум причинам. Во-первых, она помогает избежать локальной оптимизации, о которой будет сказано чуть позже. Во-вторых, понимание целевого состояния позволяет запустить механизм совершенствования, максимально приближенный к реальности, с чётким (насколько это возможно) направлением улучшений.

Собственно, упражнение по картированию потока выглядит несложным: необходимо определить ключевые шаги обработки запроса, для каждого указать суть выполняемой работы, выстроить данные шаги в последовательность получения полезного результата. Одна из возникающих трудностей — излишняя детализация блоков, когда общая схема не помещается на один лист. Авторы книг, упомянутых выше, в качестве ориентира рекомендуют ограничиться пятнадцатью блоками, иначе дальнейшая работа с картой будет осложнена. Вторая трудность — договориться участникам упражнения о том, какие же именно шаги, как и кем выполняются. В некоторых организациях отсутствует общее понимание процесса, что приводит к многочасовым спорам.

Составив схему, можно приступить к наполнению её важными подробностями. Возможно, будет полезно добавить названия задействованных исполнителей. Не лишним также станет указание мест, где накапливаются очереди объектов, ожидающих обработки, а также места, где задержки возникают по причине ожидания какого-либо календарного события — например, ежемесячной встречи по рассмотрению запросов на изменения или ежеквартального рассмотрения скорректированного бюджета. Наконец, наиболее ценная информация — три метрики для каждого шага потока, а именно: **время выпуска** (англ. Lead Time, LT), **время обработки** (англ. Process Time, PT) и **доля работ, выполненных без ошибок** (англ. Percent Complete and Accurate, %C/A). Определение значений данных метрик на практике представляет собой большую сложность для организации, не имеющей инструментов и практики измерения подобных показателей. Группа сотрудников, выполняющая картирование, может занижать временные показатели, если обсуждаемые значения кажутся ей излишне высокими. Иногда, напротив, группа может вспоминать крайние случаи, когда отдельный запрос или запросы обрабатывались слишком долго, пытаясь таким образом завысить значение времени выпуска. Ещё хуже дело обстоит с показателем %C/A, так как его значение для каждого шага, как правило, не известно и может быть лишь оценено. Важно помнить, что для составления схемы «как есть» следует опираться именно на текущее состояние дел, а не описанное в каких-либо регламентах, существующее в фантазиях руководителей или применимое лишь для исключительных случаев. Абстрактный пример карты потока создания ценности приведён на Рис. 3.5.6.

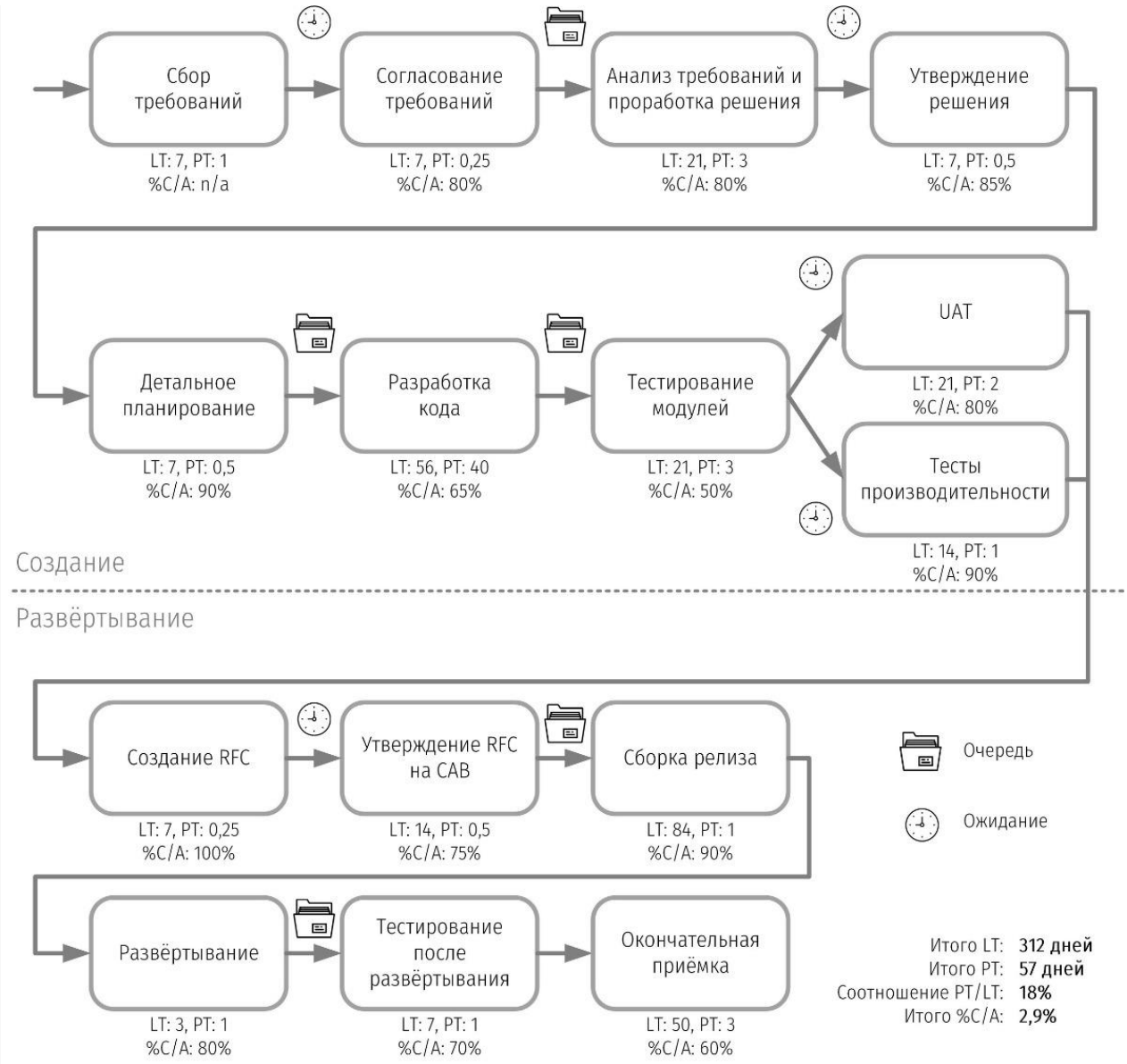

Рис. 3.5.6. Пример карты потока создания ценности.

Зачем же нужно картирование потока и почему этот поток так важен для DevOps?

Во-первых, само упражнение по созданию карты и полученные значения ключевых метрик действуют на участников процесса очень отрезвляюще. Как правило, многие понимают, что при текущей организации деятельности есть точки неэффективности, однако, никто не догадывается о масштабах бедствия, тем более в цифрах. В приведённом выше примере соотношение продуктивного времени, потраченного на получение полезного результата (создание ценности), составляет лишь 18% от общего затраченного календарного времени. Данное значение приведено не в отрыве от реальности — в обычных ИТ-подразделениях получаются примерно такие числа. Ещё хуже дело обстоит с показателем %C/A, если в организации есть привычка отправлять обратно на предыдущие шаги задачу, которая была сделана не точно, не до конца или не в соответствии с заданием.

Во-вторых, наглядное представление деятельности позволяет концентрироваться на создаваемой ценности, а не на выполняемой работе. Сотрудникам и руководителям намного заметнее и понятнее ежедневные задачи, которые они решают (ответ на вопрос «что?»), в то время как получение полезного результата ускользает от внимания (ответ на вопрос «зачем?»).

В-третьих, карта потока создания ценности даёт возможность искать и устранять узкие места, избегая при этом ловушки локальной оптимизации — расходов времени и усилий по устранению затруднений, которые не дадут эффекта вовсе, либо полученный эффект будет незначительным. В соответствии с теорией ограничений, предложенной Илияху Голдратом (https://tocinstitute.org/theory-of-constraints.html), в любой системе в один момент времени есть одно и только одно действительно узкое место, замедляющее работу, и усилия, потраченные не на его устранение — потрачены впустую. Таким образом, с потоком можно работать как с единой системой. Очевидные вопросы после выполнения картирования, которые требуют осмысления, анализа и действий, следующие:

1. **[%C/A]** Почему на участках работы получены значения %C/A, отличные от 100%, и каким образом можно добиться полного отсутствия ошибок при передаче работы с одного участка на другой (и, таким образом, потерь времени и ресурсов на переделку работы)?
2. **[LT]** На что именно расходуется время выпуска, помимо создания полезного результата, и каким образом можно радикально уменьшить простои в очередях и ожидании?
3. **[PT]** Какие есть возможности изменения практик работы, позволяющие уменьшить время обработки на каждом из участков?

Следует отметить, что подобная работа по оптимизации не должна сводиться исключительно к анализу карты «как есть» и попыткам улучшения связанных с ней метрик. Напротив, необходима разработка карты «как будет», возможно, принципиально отличающейся от текущей схемы работы. Именно здесь появляются возможности применения инструментов и практик DevOps, изменяющих существующее положение вещей.

И, наконец, в-четвёртых, осознание потока создания ценности позволяет реализовать одну из основных идей DevOps — плавное и равномерное течение (англ. Flow) работы от участка к участку, позволяющее создавать результаты постоянно, ритмично, без лишних задержек и с оптимальной загрузкой ресурсов.

Конвейер развёртывания

Понимание потока создания ценности — необходимый и важный шаг на пути к DevOps. Однако, работа с потоком «на бумаге» не даёт столь значительных результатов, как ожидается. Необходимость построения чего-то, подобного конвейеру, наглядно иллюстрируется следующим примером: засеките время, которое необходимо для того, чтобы эффект от одной новой строчки программного

кода в любом из ваших приложений появился в среде эксплуатации. Если измерения покажут результат в днях, неделях или месяцах — ваш поток создания ценности нуждается в серьёзном пересмотре. Помочь такому пересмотру призван конвейер развёртывания, под которым понимается максимально автоматизированное сопровождение изменения по всем шагам потока создания ценности, начиная с момента «Разработка завершена» вплоть до состояния «Развёрнуто в среде эксплуатации».

Работа конвейера может быть проиллюстрирована схемой (Рис. 3.5.7).

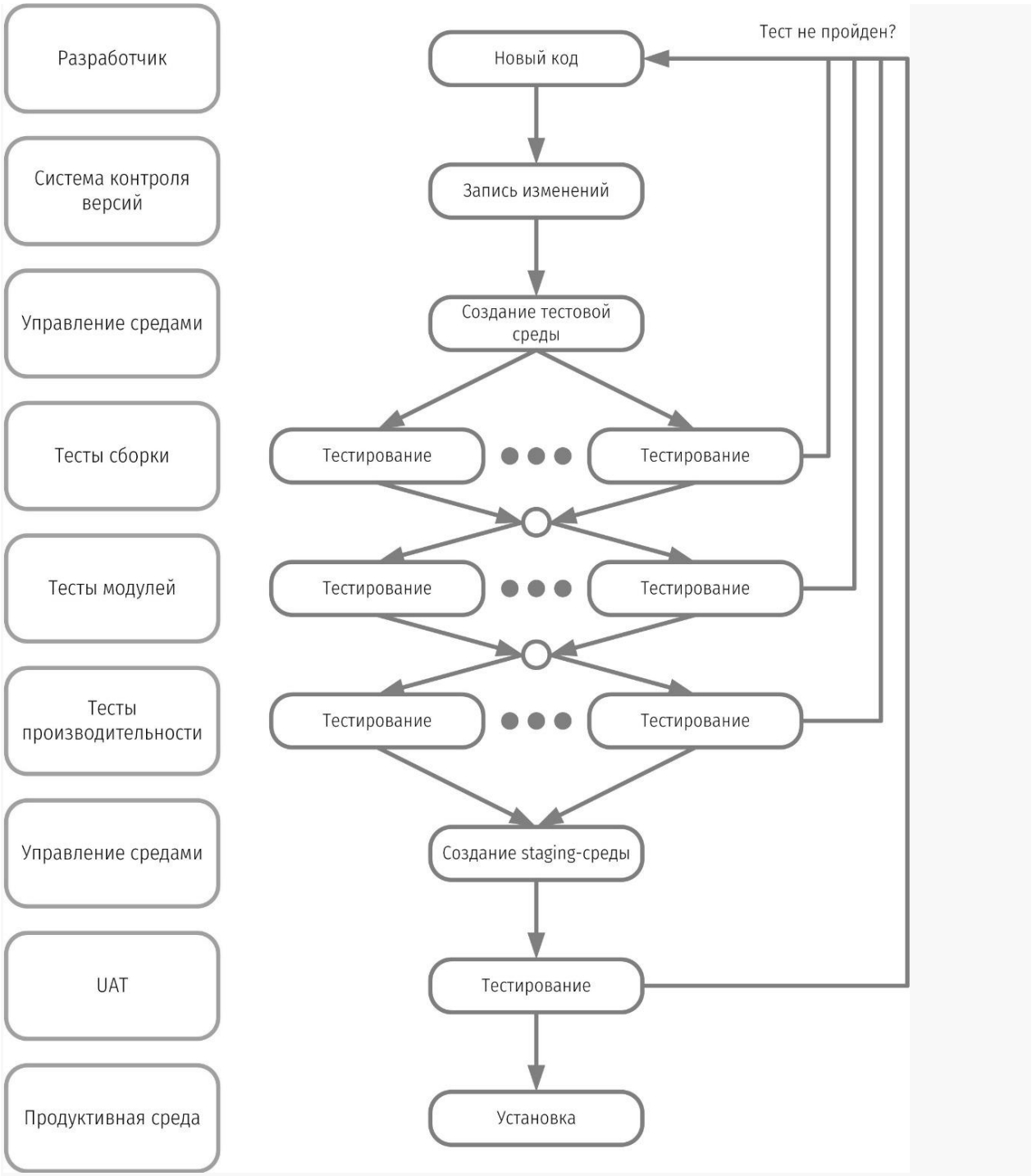

Рис. 3.5.7. Работа конвейера развёртывания.

Конвейер автоматически запускается после того, как разработчик разместит в системе контроля версий новую часть программного кода, при этом фиксируется: кто, когда и какое изменение внёс. По факту, новой записи автоматически создаётся необходимая временная тестовая среда, в которой последовательно запускаются заранее разработанные тесты. Логика размещение тестов проста: проверки, обеспечивающие выявление большинства потенциальных ошибок, располагаются максимально ближе к началу конвейера. Все тесты, требующие ручного труда (если таковые имеются), размещаются в конце конвейера. Невозможность прохождения какого-либо теста приводит к предоставлению разработчику обратной связи и остановке конвейера для данного изменения. Чтобы запустить конвейер вновь, разработчик должен исправить программный код. Помимо создания тестовой среды, возможно автоматическое создание других необходимых для конвейера сред. После использования ресурсы, занятые под эти среды, автоматически освобождаются. Разумеется, возможно параллельное исполнение нескольких тестов, если это допускается логикой тестирования и исключает непродуктивную загрузку ресурсов на тестирование изменений, которые могли бы быть отброшены на предыдущих шагах конвейера.

Таким образом, конвейер позволяет решить четыре важные для DevOps задачи. Во-первых, конвейер экономит ресурсы, не задействуя следующие шаги при непрохождении предыдущих. Во-вторых, конвейер обеспечивает качество продукта — изменения, нарушающие функциональность, не доходят до установки в среду эксплуатации, и система всегда находится в рабочем состоянии (об этом будет дополнительно сказано позже). Под качеством в данном случае понимаются все вопросы, связанные с функциональностью, производительностью, доступностью, безопасностью и т.д. В-третьих, конвейер ускоряет доставку изменений до среды эксплуатации за счёт максимально возможной автоматизации каждого из шагов. Наконец, в-четвёртых, работа конвейера постоянно «оставляет следы» в журналах аудита, что позволяет обеспечить контроль всех проводимых изменений, а также снимать точные измерения на разных участках работы конвейера, предоставляя ценные данные для его оптимизации.

С построением эффективно работающего конвейера развёртывания на практике возникают следующие сложности:

1. Чрезмерное увлечение автоматизацией в ущерб идеологии (процессы, персонал, культура) приводит к созданию замечательно автоматизированных конвейеров, которыми никто не будет пользоваться. Решение очевидно: DevOps — это не только автоматизация, и это должен понимать каждый участник команды.
2. В исходном состоянии для устойчивой работы конвейера нет достаточного количества разработанных ранее тестов. В такой ситуации иного решения, как увеличивать покрытие кода тестами быть не может — накопленный технический долг придётся рано или поздно выплачивать.
3. В целевом состоянии тестов становится так много, что прохождение изменения по конвейеру занимает очень долгое время и требует очень больших

вычислительных ресурсов, что особенно актуально при большом потоке небольших изменений. Компании, столкнувшиеся с данной проблемой, активно применяют **анализ влияния тестирования** (англ. Test Impact Analysis). За немного некорректным, но уже устоявшимся названием скрывается практика, при которой по особым меткам, а также с использованием средств искусственного интеллекта, система тестирования выбирает из всего многообразия тестов те, которые относятся к предлагаемому изменению, не выполняя оставшиеся тесты.

С построением конвейера развёртывания связаны ещё три важных для DevOps понятия: непрерывная интеграция, непрерывная поставка и непрерывное развёртывание (англ. Continuous Integration, Continuous Delivery и Continuous Deployment). Существуют их разные трактовки; следующее далее описание опирается на точку зрения экспертов, стоявших у истоков данных понятий.

Под **непрерывной интеграцией** принято понимать процесс постоянной сборки программного кода; «непрерывно» же означает, что сборка производится каждый раз, когда какой-либо разработчик размещает очередное изменение в системе контроля версий. Обычная практика разработки программного обеспечения подразумевает множество отдельных веток программного кода, в которых различные программисты и команды довольно продолжительное время (дни, недели и месяцы) трудятся над созданием новой функциональности. По завершении своей части разработки, или, что ещё хуже, после ожидания, когда все команды, работающие над одним продуктом, завершат разработку, начинается болезненный процесс сборки всех наработок в единую базу программного кода. Так как программистов много, работают они в целом асинхронно, каждый над крупными изменениями, да ещё и долгое время, то процесс сборки сам по себе является трудоёмкой задачей, занимающей несколько недель. Действительно, необходимо учесть все изменения, сопоставить их друг с другом, обновить тесты с учётом изменений и сопоставления, переписать частично или полностью некоторую уже разработанную функциональность, и всё это повторять до тех пор, пока новый код не будет приведён в рабочее состояние. Сборка — важный этап разработки ПО, являющийся, по сути, первым тестом. От того, случилась сборка или нет, зависят дальнейшие работы.

Непрерывная интеграция, впервые описанная в книге К. Бека «Объясняем экстремальное программирование», заключается в упрощении сборки и превращении её в рутину. Ожидается, что программисты будут работать в минимальном числе веток, в идеале — в общей единой базе программного кода. Также подразумевается, что разработчики вносят минимальные изменения, порционно, каждое из которых несёт небольшой риск, но тут же запускает процесс сборки — таким образом, каждый программист размещает свои наработки в системе контроля версий минимум один раз в день. Первичное тестирование, выполняемое автоматически при каждой сборке, позволяет сразу же выявить ошибки и исправить их незамедлительно, что позволяет поддерживать систему всегда в рабочем состоянии.

Непрерывная поставка, подробно описанная Д. Хамблом в одноимённой книге, расширяет идею непрерывной интеграции: каждое сохранение изменений программного кода в системе контроля версий запускает не только процесс сборки, но весь конвейер развёртывания. Таким образом, все изменения, не прошедшие полное тестирование, не принимаются и требуют немедленного исправления. А все безошибочные изменения приводят систему к состоянию полной готовности развёртывания в среду эксплуатации.

Непрерывное развёртывание заключается в переходе от состояния «система всегда готова к развёртыванию с учётом всех выполненных изменений» к состоянию «любое изменение незамедлительно разворачивается в среде эксплуатации». Переход к непрерывному развёртыванию, в частности, требует переопределения понятия «релиз»: теперь не ИТ-, а бизнес-подразделение решает, когда будет доступна та или иная функциональность. Технически, функциональность уже присутствует в среде эксплуатации, сразу по факту завершения разработки и тестирования, но её активация может быть выполнена дополнительно через программные настройки, или флаги, тогда, когда это будет нужно, скажем, отделу маркетинга. Такая практика работы называется теневыми релизами (англ. Shadow Release) или тёмными запусками (англ. Dark Launches).

В любом случае, в основе данных практик — тот самый конвейер развёртывания, описанный выше.

Всё должно храниться в системе контроля версий

Современных разработчиков программного обеспечения не удивить системами контроля версий. Первые такие инструменты, называвшиеся системами хранения исходного кода, появились ещё в 1970-х годах. Сегодня сложно встретить программиста, не знакомого с Git, Subversion или Mercurial. Да что программисты — многие web-мастеры размещают в таких системах не только исходный код, но и копии среды эксплуатации, например, для интерпретируемых Интернет-систем или web-сайтов.

DevOps, как и во многих других областях, расширяет применение таких систем. Речь идёт о хранении не только исходного кода, но абсолютно всего, связанного с ИТ-системой: тестов, скриптов создания и модификации баз данных, скриптов сборки, скриптов создания сред (включая среду разработки), скриптов развёртывания, артефактов, используемых библиотек, создаваемой документации, конфигурационных файлов, даже средств разработки, компиляторов, IDE и прочих инструментов. Перед каждым элементом приведённого списка уместно поставить дополнение «всех»: всех тестов, всех скриптов и так далее. Исключение делается только для двоичного кода, являющегося результатом компиляции программы, по следующим соображениям: обычно код занимает значительное место (что существенно, если он пересоздаётся после каждого изменения) и может быть воссоздан при наличии в системе хранения версий всего остального.

Данный принцип позволяет иметь беспрецедентный уровень контроля за всеми составляющими частями работающей системы, недостижимый при использовании других инструментов. Разумеется, применение такого принципа требует изменения культуры работы с информацией и конфигурациями.

Одним из следствий его применения является возможность установить: что, когда и кем было изменено. Другая важная возможность — способность восстановить систему на любой момент времени в прошлом, в том числе вернуть «сломанную» систему в гарантировано рабочее состояние с минимальными трудозатратами.

Автоматизированное управление конфигурациями

Развивая далее принцип, описанный в предыдущем разделе, DevOps полностью перестраивает работу со средой эксплуатации (впрочем, равно как и с любыми другими средами). Традиционная практика многих компаний такова: новый сервер создаётся из заранее подготовленного образа, затем администратор вручную производит его настройку, устанавливая и конфигурируя дополнительные пакеты программного обеспечения, как системные, так и прикладные. В случае необходимости изменения состава пакетов или их конфигураций, администратор под своей учётной записью подключается к серверу и вручную производит необходимые настройки.

В мире DevOps такая практика работы полностью исключена: любые изменения любой среды могут выполняться только скриптами, располагающимися в системе контроля версий. Например, если с завтрашнего дня в тестовой среде необходимо иметь новую библиотеку, то администратор должен исправить скрипт создания тестовой среды, протестировать его работу и разместить его в системе контроля версий. Создание сред выполняется автоматически при работе конвейера развёртывания.

Многие описанные ранее отличия DevOps от обычной практики касались, в первую очередь, разработки и тестирования, и лишь иногда затрагивали интересы эксплуатации. Этот же принцип требует полной перестройки практики работы отделов ИТ-поддержки и ИТ-сопровождения. Действительно, теперь администраторы не имеют права что-то менять в среде эксплуатации, за которую они отвечают, привычными им способами.

При использовании управления конфигурациями по DevOps получаются те же преимущества, что и от контроля версий, но в первую очередь — для ответственных за эксплуатацию. Теперь все изменения контролируются, систему можно быстро восстановить до рабочего состояния, знания с уходом ключевого персонала не будут утеряны, и так далее.

Некоторые апологеты DevOps настолько рьяно защищают такую практику работы, что предлагают устанавливать и тщательно настраивать системы тотального аудита ИТ-инфраструктуры для выявления несанкционированных изменений на

любом участке с последующим немедленным увольнением персонала, позволившего себе вручную настроить какой-либо сервер или элемент сети. Для небольшого и среднего размера компаний, возможно, данная практика выглядит чрезмерной, однако, если у вас тысячи серверов и сотни инженеров, то другого пути обеспечения стабильности, качества и скорости может и не найтись.

Отдельные команды идут ещё дальше: автоматизированные системы регулярно изменяют административные пароли для доступа к разным средам, не сообщая новые пароли ИТ-сотрудникам. Таким образом обеспечивается отсутствие несанкционированных изменений в среде эксплуатации, хотя это правило действует для любых сред: разработки, тестирования, стабилизации и пр.

Определение завершения

Традиционное отношение любого обычного сотрудника к выполняемой работе можно условно обозначить фразой: «Я свою работу сделал, я молодец». Действительно, именно за выполнение своей трудовой функции сотрудник и получает заработную плату. Аналитик разработал функциональные требования — его работа завершена. Разработчик написал программный код — выполнил свою часть общего дела. Тестировщик протестировал — завершил свою часть, и так далее. Однако, в DevOps всё совсем не так.

Один из ключевых принципов: работа завершена не тогда, когда кто-то сделал свой объём, а когда заказчик получил или начал получать ту ценность, на которую рассчитывал. Это означает полное прохождение всего потока создания ценности вплоть до среды эксплуатации, только тогда работа будет считаться завершённой.

При достаточной очевидности данного принципа, следование ему не появляется само собой и требует управленческих усилий. Такие усилия в дальнейшем позволяют получить следующие преимущества:

1. Команда фокусируется не на выполнении работы (что делаем), а на результатах, ценности для клиента (зачем делаем).
2. Ограниченная ответственность за от-дельные участки работы («к пуговицам претензий нет?») размывается, заменяясь коллективной ответственностью за общий результат команды («костюм должен сидеть»).

Радикально настроенные адепты DevOps настаивают на более жёстком определении завершения. Они предлагают использовать принцип, при котором создание новой функциональности завершено тогда, когда приложение работает в среде эксплуатации и все действия по сборке, тестированию и развёртыванию выполнены автоматически.

Обзор ключевых отличий DevOps-практик от традиционных

Сравнение DevOps-практик с условными «традиционными» практиками через подчёркивание отличий поможет уловить самое важное.

Релиз — это рутина

В обычной работе ИТ-отдела каждый релиз — это большая проблема. В релиз, как правило, включается множество изменений, связанных со множеством запросов заказчиков. Туда же добавляются изменения со стороны самого ИТ-отдела — то, что необходимо сделать, чтобы системы продолжали работать или работали ещё лучше (стабильнее, безопаснее, быстрее и так далее). Проверить такой большой релиз — отдельная задача, требующая внимательности, времени, привлечения множества специалистов. Все знают, что для любого релиза что-то обязательно пойдёт не так, поэтому ИТ-сотрудники:

- разрабатывают специальные документы, описывающие изменения (забывая при этом часть из них);
- готовят дополнительные резервные копии (для больших систем занимающие много места и долгое время, создавая дополнительную нагрузку на системы и сети, и всё равно кто-то забудет положить в них важные файлы);
- планируют специальные действия и разрабатывают пошаговые инструкции по возвращению системы, если это возможно, в исходное состояние, когда что-то пойдёт не так (особенно интересны случаи, когда релиз частично «установился», а частично — нет);
- ищут время в согласованном календаре изменений, позволяющем остановить работу системы — планово, если всё пройдёт хорошо, либо экстренно, если что-то пойдёт не так (такое время обычно находится в ночь с пятницы на понедельник);
- только после этого распространяют релиз, выполняя довольно большое число действий вручную (и не фиксируя промежуточные результаты).

В зависимости от тщательности проработки каждого из пунктов данного списка, длительность всего развёртывания может варьироваться от нескольких дней до нескольких недель. Количество бессонных ночей администраторов и разработчиков зависит от размера релиза, состояния ИТ-системы и усилий по подготовке и распространению релиза.

В DevOps релиз — это рутина. Релизы выполняются еженедельно, а то и ежедневно. Разумеется, для этого необходимо кардинально уменьшить размер вносимых изменений, но не только: также необходимо самым радикальным образом пересмотреть практику выполнения работ по подготовке и распространению релизов. Вспомним конвейер и практики непрерывной

интеграции и непрерывной поставки — они позволяют документировать все изменения в системе контроля версий, большинство операций сделать с помощью автоматизированных средств, учесть в журналах все проведённые изменения, сразу же настроить мониторинг новых и изменённых компонентов. В случае каких-либо неполадок при развёртывании конвейер автоматически прекратит распространение, откатит назад уже внесённые изменения и оповестит команду для принятия мер.

Выпуск релиза — решение бизнеса

Строго говоря, в предыдущем разделе слово «релиз» используется не совсем корректно. Дело в том, что релиз в ITSM и релиз в DevOps — понятия различные. Для классического ИТ-менеджмента релиз — совокупность нескольких изменений, распространяемых в среде эксплуатации совместно. В то время как в DevOps релиз — это включение новой функциональности, чтобы она полностью или частично стала доступна пользователям. Более правильно в предыдущем разделе вместо слова «релиз» применительно к DevOps использовать слово «поставка», однако, оно ещё не так хорошо вошло в русскую речь, потребуется ещё несколько лет, чтобы сделать его таким же привычным, как «релиз».

Итак, в обычной работе релиз — это решение ИТ-департамента. Есть некий календарь или политика релизов, определяющие возможные частоту и масштаб, и даже нумерацию версий. Бизнес-подразделение, которому необходимо получить новую функциональность для своих клиентов, встаёт в очередь и дожидается очередного релиза: в счастливом случае ближайшего, но зачастую — подальше, через один-два квартала.

При использовании непрерывного развёртывания в DevOps поставка новой функциональности в среду эксплуатации производится сразу же, как она разработана и протестирована. Пользователи её не замечают, так как она пока не активирована. Активация выполняется тогда, когда это необходимо бизнес-подразделению в соответствии с его маркетинговыми, рекламными или иными планами и соображениями. Такая практика не только позволяет передать управление релизами в руки заказчика, но и получить дополнительные преимущества.

Во-первых, радикально сокращается, вплоть до исчезновения, **время простоя при распространении релизов** (англ. Zero-Downtime Releases). Во-вторых, появляется возможность выполнять **сине-зелёные развёртывания** (англ. Blue-Green Deployments), для которых создаются две копии среды эксплуатации: «зелёная» и «синяя», соответственно. Переключение пользователей с одной среды, где они пока взаимодействуют с предыдущей версией приложения, на другую, где уже подготовлена новая версия, производится менее, чем за секунду. В-третьих, компании с большим числом пользователей могут использовать технику так называемых **канареечных релизов** (англ. Canary Releases), когда новая функциональность сначала становится доступной небольшому подмножеству

пользователей. Убедившись, что всё в порядке с технической и с маркетинговой точек зрения, может быть принято решение о переключении всех остальных пользователей, при этом, первоначальная сегментация выполняется бизнес-подразделениями по той логике, которая им важна и близка: по территориальному признаку, тарифным планам клиентов, лояльности клиентов или иным. Наконец, в-четвёртых, многие компании начинают активно применять A/B-тестирование для проверки бизнес-гипотез, когда часть пользователей (контрольная группа) работает со старой версией системы, а другая часть (экспериментальная группа) использует уже новую версию. Измерение ключевых показателей и сравнение групп между собой позволяет бизнесу проверять свои идеи и корректировать дальнейшее развитие данной системы.

Всё перечисленное становится возможным только если изменяется сама суть релиза, и решение передаётся в руки бизнеса.

Автоматизируется всё, что только возможно

Известная пословица «Лень — двигатель прогресса» применительно к ИТ трансформируется в наблюдение «Ленивый администратор в конце концов напишет скрипт, чтобы меньше работать». В традиционном ИТ-отделе ждать написания скриптов можно долго, единого хранилища нет, их работоспособность остаётся под вопросом, поэтому большинство операций, в том числе часто повторяемых, выполняется вручную. Среди них необходимо отдельно отметить:

- создание сред (тестирования, промежуточных и иных);
- конфигурирование элементов инфраструктуры;
- тестирование;
- развёртывание и тиражирование, включая настройку средств мониторинга.

Важное для DevOps повышение уровня контроля требует тотальной автоматизации всех ручных операций, в особенности — перечисленных выше. Необходимые для работы конвейера развёртывания среды создаются скриптами и автоматически под управлением системы управления конвейером. Также автоматически эти среды уничтожаются после использования, освобождая ресурсы. Быстрая работа конвейера требует максимальной автоматизации всего тестирования, насколько это возможно. Ручные тесты остаются на самый крайний случай, хотя новые достижения постоянно сдвигают границу такого случая: сегодня можно выполнять автоматическое тестирование не только модулей, интеграции, регресса, функциональности, производительности, но и пользовательского интерфейса, удобства использования, приёмочных испытаний. Развёртывание и тиражирование, как завершающие шаги конвейера, также выполняются автоматически, с необходимой подстройкой средств мониторинга систем и приложений. Данный шаг нельзя недооценивать — качественно настроенный мониторинг позволяет получать очень быструю обратную связь относительно новых релизов. Как бы сотрудники не старались приблизить конфигурацию тестовой среды к среде эксплуатации, разница может проявиться уже после

развёртывания. В таком случае событие, зафиксированное системой мониторинга, может привести к автоматическому откату назад уже развёрнутого изменения для обеспечения стабильности среды и приложений.

Более того, при переходе от традиционных монолитных архитектур к микросервисным полный мониторинг компонентов становится насущной необходимостью, ведь это единственная возможность отследить не только работоспособность, но и фактическое использование данного сервиса или данной версии сервиса другими сервисами. Без такого контроля эволюционирующая архитектура не сможет развиваться, и в ней будут постоянно накапливаться уже умершие, но всё ещё не отключённые сервисы.

Устранение сбоев не подразумевает очереди

Типичный процесс управления сервисными инцидентами, когда о случившемся сбое сообщает пользователь, устроен так:

- пользователь обращается на первую линию поддержки через телефон, электронную почту, портал, онлайн-чат или мобильное приложение;
- первая линия поддержки (с помощью сотрудника, автоматизированной системы или средств искусственного интеллекта) регистрирует и классифицирует обращение, в том числе присваивая ему приоритет, влияющий на скорость дальнейшей обработки;
- обращение попадает в очередь, где ожидает своего часа (или дня).

Управление инфраструктурными инцидентами, когда информация о сбое поступает от ИТ-специалиста или системы мониторинга, устроено примерно так же, завершаясь очередью. Наличие очереди — механизм управления, связанный как с необходимостью упорядочивания работы, попыткой более равномерно загружать ресурсы, но ещё — с длительным временем решения инцидентов. Для каждого инцидента необходимо произвести расследование, выполнить диагностику, найти и применить обходное решение — всё это в подавляющем большинстве случаев выполняется вручную.

В идеальном DevOps всё не так. В случае, если инцидент связан с развёртыванием, которое недавно состоялось (то есть можно отследить причинно-следственную связь), система управления конвейером автоматически выполнит возврат к предыдущему известному рабочему состоянию. Вмешательство человека требуется для анализа проводимого изменения и его корректировки, что выполнить намного легче и быстрее, ведь данное изменение было совсем недавно, а не несколько месяцев или лет назад. Известны решаемая задача, заказчик, разработчик, тестировщик — все участники цепочки.

В том случае, если что-то «сломалось» в инфраструктуре, принимается решение без долгих разбирательств отключить сбойный элемент (например, сервер приложений) и создать этот участок инфраструктуры заново, пользуясь уже

готовыми и отлаженными скриптами, с помощью которых элемент был создан ранее. Такая операция занимает намного меньше времени, чем при обычном процессе. Действительно, если под управлением ИТ-отдела находится, скажем, несколько десятков серверов, то можно каждый из них настраивать вручную, придумывать ему уникальное красивое имя, холить и лелеять. Но когда ИТ-подразделение управляет сотнями и тысячами серверов, такой способ вносит слишком большие ограничения и не является продуктивным. Альтернативный подход DevOps зачастую называется «**стадо, а не домашние любимцы**» (англ. Pets vs. Cattle). Напомним, что DevOps подразумевает максимальное абстрагирование от реального аппаратного обеспечения в пользу виртуализации, что было описано ранее.

Дефекты исправляются немедленно

В работе обычного ИТ-отдела выявленные при эксплуатации ошибки, которые каким-то образом прошли через тестирование, оцениваются, приоритизируются и встают в очередь. В самой описанной процедуре нет ничего негативного, кроме того факта, что многие из ошибок встают в очередь навечно, накапливая таким образом технический долг. Присвоив незначительный приоритет, команда откладывает устранение такой ошибки на длительный срок. К моменту, когда срок подходит, во-первых, все давно забыли, что за ошибка, почему происходит и как её устранить, а во-вторых, находится более важная и срочная работа. Первое требует восстановления контекста и дополнительных трудозатрат, второе делает невозможным устранение неприоритетных ошибок при наличии более приоритетной работы, которая, как правило, всегда есть.

Ещё одна сложность, наблюдаемая на практике, заключается в невозможности объективно оценить размер очереди ошибок, которая имеет тенденцию к разрастанию. Десять ошибок — это ещё допустимо? А пятьдесят? Пятьсот? Как сравнить ошибки разных приоритетов, значимости или ущерба? Может ли ошибка, находящаяся в очереди неделю, подождать ещё? А месяц? Год? Принимая во внимание, что очередь ошибок находится в недрах какой-либо системы учёта, увидеть и осознать её — уже проблема. Особенно если к картине добавить аргументы вроде «Эту ошибку устранять уже нет смысла, так как данный модуль через полгода планируем менять на другой». Для реалистичности картины нужно обязательно указать, что ошибка находится в очереди уже не меньше года, а «планируем» вовсе не означает «заменим». И, как правило, не через полгода.

В DevOps исправление ошибок устроено иначе. В соответствии с принципом «система должна всегда находиться в рабочем состоянии», а также, в стремлении управлять техническим долгом, большинство выявленных ошибок получают приоритет, требующий немедленного устранения — например, в рамках того же или ближайшего спринта, если команда работает по Scrum. В случае выявленных минорных ошибок допускается выделение увеличенного срока на устранение, однако он должен быть не слишком большим и в любом случае должен быть соблюдён.

Как и многие другие практики, такой порядок означает большую перестройку в планировании, приоритезации и выполнении работ, а кроме того — серьёзные изменения в исходных принципах организации процессов. Многие руководители просто не согласятся с принципом «выявленные ошибки устраняем немедленно». Возможно, как ранее не соглашались с принципом из ITSM: «Все поступившие заявки должны быть зарегистрированы». В этом случае один из методов — работать с выявленными дефектами так же, как производится работа над новой функциональностью. Ошибки и пользовательские истории попадают в единую очередь и рассматриваются на равных. Действительно, выбор между реализацией той или иной возможности делается по тем же исходным основаниям, как и выбор — какую ошибку устранять. Равно как и предоставление предпочтения разработке новой функциональности в ущерб устранению дефектов — такое же управленческое решение, принимаемой для той же ИТ-системы, тех же ресурсов, тех же пользователей. В этом случае к управлению техническим долгом привлекаются заказчики, что сильно меняет как значимость такой работы, так и ответственность за её результаты.

Процесс улучшается постоянно

Ещё хуже в обычном ИТ-подразделении обстоит дело с изменением процесса работы. Консультанты, рабочая группа, состоящая из сотрудников компании, а то и специализированное подразделение разработали необходимые регламенты. Как правило, они описывают некую модель, в разной степени соответствующую желаемому порядку выполнения работ: как и любая модель, данные регламенты будут содержать разрыв между желаемой практикой и описанием. Например, сложно предусмотреть все возможные ситуации и отклонения, сложно описать мотивировочную часть — зачем и почему такая работа должна выполняться таким способом, сложно детализировать изложение до необходимого уровня, при этом никого не запутав и не превратив сотрудников в роботов. Следующий разрыв между реальностью и регламентов возникает, когда реальное выполнение работ становится не таким, как предполагалось. Где-то сотрудники будут срезать углы, где-то стараться работать лучше, чем диктуют инструкции. Третий разрыв связан с автоматизацией оперативных процессов, от которой эти процессы сильно зависят. Во многих случаях настройка инструмента автоматизации производится с большими задержками относительно выполнения процесса: работа уже выполняется иначе, но система автоматизации пока не изменилась. Либо, что ещё хуже, работа выполняется не оптимальным образом потому, что нет возможности оперативного изменения системы автоматизации. В одной известной мне организации очередь на внесение изменений в ITSM-систему составляет два года, что сильно замедляет реализацию всех назревших корректировок процессов.

Такое большое количество разрывов крайне негативно влияет на практику выполнения работ. Поэтому в DevOps используется иное правило: любые выявленные недостатки процесса должны быть устранены немедленно. Например, если некорректно работает какой-либо скрипт, обеспечивающий работу конвейера

развёртывания, его нужно незамедлительно исправить. Более того, в противовес традиционной практике, при которой проблемы можно отложить, в DevOps рекомендуется проблемные шаги повторять как можно чаще. Это позволит лучше понять, как именно их следует улучшить, исправить, и внести соответствующие корректировки в работу.

Стартап как ориентир

Некоторые DevOps-команды возникли в стартапах, с их необычной культурой, такой непривычной для корпоративных сотрудников. Компании, пытающиеся выстраивать DevOps у себя, стараются перенять этот дух предпринимательства и инноваций. Но что же это означает? В чём состоит разница, можно ли её сформулировать? Оказывается, можно — Табл. 3.5.2 перечисляет ключевые отличия.

Табл. 3.5.2. Отличия в культуре обычных корпораций и стартапов.

Характеристика	Культура обычных корпораций	Культура стартапов
Стиль управления	Командный, авторитарный	Автономный
Склонность к переменам	Консервативность	Эксперименты
Организационная структура	Функционально-иерархическая	Сетевая
Акцент результата	Проектно-ориентирован	Ориентирован на продукт
Модель	Водопадная	Гибкая, итеративная
Системная архитектура	Монолитная, тщательно спроектированная	Слабо связанная, микросервисная
Предпочтения в технологиях	Патентованные, проприетарные	Открытый исходный код

Похоже, по всем приведённым характеристикам DevOps-культура сильно отличается от привычной, что, конечно, препятствует прямому и быстрому изменению стиля работы в обычных корпорациях. Приведённая выше таблица хорошо суммирует основные различия, позволяя перейти к более детальному рассмотрению отдельных DevOps-практик. Напомним, что многие из них являются, условно говоря, заимствованиями из других известных областей, что не умаляет значимости как этих областей, так и DevOps.

Необычные команды

В таблице предыдущего раздела, в колонке «Культура стартапов» были приведены некоторые отличия, делающими невозможным, либо крайне затруднённым использование традиционного функционального управления. В частности,

автономный стиль, ориентация на продукт и сетевая организационная структура подталкивают к пересмотру способа группировки специалистов для достижения больших результатов. На первый план выходят команды, а не структурные подразделения.

DevOps-команда — удивительная боевая единица. Она отвечает за небольшой, но чётко обозначенный кусочек какой-либо ИТ-системы или ИТ-инфраструктуры. Обладая строгим фокусом, члены команды постепенно и неизбежно становятся экспертами в данной предметной области, сохраняя полную ответственность за неё.

Команда не является способом объединить сотрудников на время, например, на проект — напротив, команда создаётся на долгий срок. Более того, как правило срок жизни команды заранее не определяется и не фиксируется. Команда работает над своей областью ответственности до тех пор, пока область остаётся релевантной. В случае изменения траектории команда «поворачивает» вместе с областью ответственности; в случае отказа от данной области команда переключается на другую. Среди практиков нет устоявшегося мнения стоит ли время от времени разрушать команды. С одной стороны, распределение участников одной, хорошо поработавшей команды между другими позволяет ускорить обмен компетенциями и опытом. Однако многие эксперты возражают, что время и ресурсы, потраченные на создание эффективной сложившейся команды лучше реинвестировать в другие задачи, сохраняя миниколлектив, а обмен знаниями можно и нужно организовывать независимо от формирования команд, и другими способами.

Участники команды работают в ней 100% своего рабочего времени: никакого больше разделения ресурсов, совмещения обязанностей там и тут, замены болеющего сотрудника в другом отделе и прочего. Полное погружение каждого участника упрощает координацию работ, убирает зависимости от внешних факторов и исключает возможность сослаться на другую загрузку. С другой стороны, такой подход увеличивает расходы на персонал.

DevOps-команда является кроссфункциональной — это означает, что она должна быть способна полностью выполнять всю работу в потоке создания ценности своей области ответственности. Только такой подход обеспечивает возможность единого и точного понимания определения завершения, только так можно доводить задачи до конца и полностью избавиться от незавершённой работы.

Размер команды имеет важное значение. С одной стороны, её не получится сделать слишком маленькой — небольшая команда не сможет стать кроссфункциональной, как описано выше. С другой, команды из двадцати и более человек сложны в координации и будут либо требовать создания уровней управления, либо будут склонны разваливаться на составные подкоманды. Кроме того, в больших командах возникают дополнительные расходы на коммуникации и неизбежная потеря информации между участниками. Всё это негативно сказывается на скорости работы.

Небольшой размер и необходимость кроссфункциональности выдвигают дополнительное требование к DevOps-командам: сотрудники должны быть максимально универсальными. Чёткая специализация привычна: вот это — программист, а это — тестировщик, а вот это — специалист по информационной безопасности. Но DevOps-команда требует стирания границ: в идеале каждый должен быть способен выполнять работу каждого. Такая особенность не означает, что все станут одинаково плохими, к примеру, разработчиками или администраторами баз данных. Понятно, что экспертиза сотрудников в отдельных областях может и должна быть глубокой. Однако универсальность позволяет членам команды помогать друг другу, обмениваться компетенциями, на экспертном уровне понимать, как всё устроено. Всё это выравнивает загрузку и создаёт единую ответственность команды как боевой единицы, а не отдельных гуру.

Среди небольшого числа участников DevOps-команды нет формального руководителя, нет координатора или супервайзора. Команда должна быть способна самостоятельно решать все возникающие управленческие вопросы, а в сложных случаях — обращаться за поддержкой к экспертам или наставникам. Проводя аналогию со Scrum, владелец продукта не обладает голосом бóльшим, чем любой другой член команды, а Scrum-мастер не является специально выделенным человеком — это всего лишь роль, время от времени передаваемая от одного участника другому. Иначе говоря, команда должна быть самоорганизующейся, что вполне достижимо для команд небольшого размера.

Важно, чтобы все члены команды физически располагались рядом, в одном помещении. Необходим постоянный личный контакт, не на расстоянии и не только через электронные средства коммуникации. Такое строгое требование имеет серьёзные основания: во-первых, коммуникации формата «напиши — прочитай» скрывают эмоциональную составляющую, независимо от способа передачи информации (электронное сообщение, мгновенное сообщение, формальный документ), точности формулировок и наличия смайликов. В совершенно очевидных случаях получателю понятно — похвалили его или предъявили претензию, но во всех остальных ситуациях основной эмоциональный посыл отправителя остаётся за кадром. Известны случаи, когда безобидные с точки зрения написанного текста комментарии вызывали бурю негодования, а сравнение с определёнными известными персонажами воспринималось как публичное оскорбление. Хорошо, если такая реакция станет заметной сразу же! Однако стоит помнить, что многие из работающих в ИТ-отрасли специалистов являются интровертами, имеющими склонность накапливать обиды. Стоит добавить к придуманным негативным эмоциям практически безграничные технические возможности, доступ к исходному коду и в среду эксплуатации — получится взрывоопасная смесь.

Во-вторых, расположение всей команды в одном помещении делает неизбежным ежедневный контакт каждого с каждым при отсутствии физических барьеров. Сообщение электронной почты, находящееся в папке «Входящие», можно игнорировать неделями. Телефонные звонки можно просто не принимать, ссылаясь на загрузку, совещания, встречи и т.д. А на неудобные вопросы стоящего

рядом коллеги отвечать придётся сразу же: условно говоря, программисту теперь не скрыться от тестировщика, а тестировщику — от специалиста по эксплуатации. Некачественная работа, дефекты, инциденты будут не только заметны и зарегистрированы в какой-либо информационной системе, они будут также максимально оперативно устранены и исправлены, при том совместными усилиями разных членов единой команды. Характерно, что такой стиль работы группы не требует наличия руководителя, координатора или иного «разводящего».

DevOps-команда сама отвечает за используемые ею инструменты. Как и из чего строить конвейер, какие применять технологии, какие версии библиотек использовать — все подобные вопросы передаются в зону ответственности команды. Она должна быть способна оценить последствия любых проводимых изменений. Данные утверждения не исключают необходимости следования корпоративным стандартам, в том числе в области архитектуры, информационной безопасности и аудита.

Область применения и ограничения DevOps

DevOps — это один из инструментов, пусть и новых, в руках современного ИТ-менеджера. Как и прочие управленческие инструменты, он не является лекарством от всех болезней, а лучше всего подходит для решения определённых задач. Более того, как и другим подходам, ему присущи некоторые ограничения.

Начнём с того, что далеко не каждой организации в принципе следует задумываться про DevOps. Для начала, отсечём особые случаи — компании, разрабатывающие программное обеспечение (в том числе на заказ), системные интеграторы, подрядчики на разных участках ИТ-работ и чисто проектные организации. Общее для перечисленных случаев — участие лишь на ограниченном участке потока создания ценности (Рис. 3.5.8).

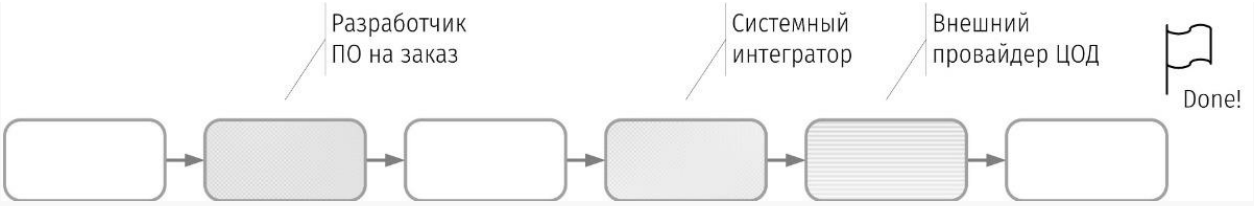

Рис. 3.5.8. Особые случаи, не рассматриваемые далее.

Изучение применимости DevOps для таких ситуаций является задачей, достойной отдельной публикации. Сфокусируемся на более традиционной компоновке: бизнес, имеющий внутреннее или внешнее ИТ-подразделение, полностью отвечающее за все вопросы применения информационных технологий. Собственно, область деятельности и форма собственности такого бизнеса не так важны — это может быть финансовое учреждение, страховая компания, торговая организация, некоммерческое партнёрство, производство товаров или предприятие сферы услуг. Главное, что в этом бизнесе используются информационные технологии, а где они — там и задачи по получению максимальной отдачи от ИТ.

Интерес к DevOps для такого рода компаний возникает при выполнении следующих условий:

- основной бизнес компании сильно зависит от информационных технологий (зависимость несложно оценить по косвенным критериям, например, по доле затрат на ИТ в общем бюджете организации и месту высшего руководителя по ИТ в иерархии управления компании);
- темп изменений, происходящих в используемых данной организацией информационных технологиях, высок;
- основной бизнес требует быстрых изменений, вызванных необходимостью проверки новых идей или гипотез;

- существуют связанные с информационными технологиями угрозы для основного бизнеса, оцениваемые владельцами или высшим менеджментом, как неприемлемые;
- все остальные испробованные способы повышения эффективности больше не дают ощутимых результатов.

Если для данной организации пункты приведённого выше списка являются релевантными, применение DevOps в том или ином виде имеет потенциальную ценность.Отдельно следует упомянуть случаи, когда компании рассматривают применение DevOps для резкого снижения накопленного технологического долга, либо устранения хрупкости ИТ-инфраструктуры. Нужно помнить, что для сложных ситуаций увлечение DevOps, скорее всего, не принесёт большой пользы и совершенно определённо не даст быстрых побед — напротив, организационные и технологические перемены могут привести к хаосу и потере остатков управляемости. Исправлять запущенные проблемы следует аккуратно, вдумчиво и рассудительно, не надеясь на DevOps как на лекарство от хронических заболеваний.

Перейдём к рассмотрению второго аспекта — реализуемости. Во всех ли компаниях можно «построить» DevOps? Мнение большого числа зарубежных экспертов сводится к положительному ответу, однако, трезвый взгляд на реальность показывает иное.

DevOps не очень подходит тем, у кого нет собственной разработки программного обеспечения — например, если всё основное применяемое ПО является уже готовым, «коробочным», и настраивается через интерфейсы взаимодействия с пользователем или администратором. Раз в компании нет собственной разработки — нет и начала потока создания ценности, нет возможности контроля версий исходного кода (так как нет доступа к исходному коду и нет необходимых компетенций с этим кодом разбираться). Зато есть существенная зависимость от компании-разработчика и от компании-поставщика применяемого программного обеспечения. Негативные следствия такой зависимости хорошо известны: какой бы крупной и известной ни была ваша организация, вы, как правило, будете лишь одним из множества заказчиков, и, несмотря на все заверения менеджеров по работе с клиентами, будете находиться в той же очереди ожидающих внимания разработчика, что и все остальные. При этом существенным является не место в очереди, а сам факт её наличия. Другое негативное следствие зависимости от внешнего разработчика «коробочного» ПО — крайняя медлительность многих компаний, производящих программное обеспечение ввиду применения ими тех самых водопадных моделей и долгих релизных циклов. Известны случаи, когда критичные дефекты в новой версии ПО остаются без исправлений более девяти месяцев, отдельные сбои «не получается» диагностировать более полугода, а клиенту предлагается безрадостный выбор: либо оставаться на устаревшей на 2-3 года версии ПО с длительной поддержкой (англ. LTS, Long Term Support), в которой вроде бы дефектов меньше, либо постоянно переходить на каждую новую версию, исправляющую предыдущие ошибки и вносящую новые.

Сложности применения DevOps возникнут и в организациях, где разработка программного обеспечения есть, но программисты выведены из штата: разработка выполняется другими компаниями на заказ, либо программисты не являются сотрудниками данной компании, а работают по договору подряда, фриланса, аутстаффинга или подобного. В таком случае полноценно включить их в поток создания ценности может быть затруднительно из-за совершенно различной мотивации участников. Сотрудники компании, находящиеся в штате, как правило более заинтересованы в удовлетворении потребностей основного бизнеса, в процветании компании, в собственном карьерном росте, а значит — в быстро полученном и качественном конечном результате работы всех участников команды. В то время как внешние разработчики могут стремиться максимально ограничить свою ответственность рамками договора и стараться выдавать результаты в строгом соответствии с полученным техническим заданием, зачастую завышая трудозатраты и перезакладываясь по срокам. К рассмотрению следует добавить возможную частую смену исполнителей, их неполное выделение в данную команду, а также типичную ситуацию, когда об объёме и условиях привлечения договариваются одни (скажем, руководитель отдела развития со стороны потребителя с менеджером по работе с клиентами со стороны подрядчика), а реально ежедневно взаимодействуют другие (собственно внешние разработчики с остальными членами команды). В описанном случае искажаются или становятся невозможными многие принципы, изложенные в разделе о командах.

Следующее ограничение применения DevOps — устоявшиеся, сложившиеся процессы, подкреплённые иерархией принятия решений, организационной структурой, внутренней нормативной документацией, бюрократией и корпоративной культурой. Некоторые крупные организации трезво оценивают свои способности меняться как ограниченные, в то время как переход к DevOps требует большой перестройки не только ИТ-отдела, но и принципов работы бизнес-подразделений. Достаточно вспомнить отличия культуры традиционных больших корпораций от культуры стартапов, приведённые ранее, чтобы оценить масштаб необходимых преобразований. Важно отметить, что для многих организаций полное изменение имеющейся практики работы является принципиально невозможным, несмотря на демонстрируемые краткосрочные успехи в отдельных частях компании.

Наконец, последним значимым препятствием является монолитная, жёстко связанная ИТ-архитектура. Организация небольших команд требует возможности закрепить за каждой из них отдельную область ответственности. В ситуации, когда рассматриваемая ИТ-система до сих пор разрабатывалась и поддерживалась десятками и сотнями сотрудников как единое целое, выделить из неё части для отдельных самостоятельных команд, работающих асинхронно, будет достаточно сложно.

К перечисленным сложностям следует добавить ещё несколько факторов, ограничивающих, по мнению многих, применение DevOps. Однако, прежде необходимо заметить, что эти факторы некорректно рассматривать как проблемы,

ставящие крест на DevOps-инициативах. Правильнее относиться к ним как к ограничениям, которые можно устранить, то есть как к задачам, имеющим решения. Вот эти дополнительные ограничения:

- Неготовность к созданию DevOps-команд. В некоторых организациях, к примеру, поощряется удалённая работа без необходимости присутствия в офисе в определённые часы. Встречаются территориально распределённые компании, где, в т.ч. и сотрудники ИТ-подразделения не находятся все в одном месте. Наконец, во многих компаниях организационная структура настолько жёсткая, что не подразумевает создания кроссфункциональных команд. Все эти примеры иллюстрируют приведённый выше тезис — они не являются стоп-фактором на пути к DevOps, они лишь требуют соответствующих изменений, корректировок, пусть непростых, но, тем не менее, возможных.
- «Особые» требования к информационной безопасности или соответствию внешним критериям. Слово «особые» намеренно взято в кавычки — более внимательное рассмотрение вопроса в конкретной компании может показать, что в действительности данная организация ничем принципиально не отличается от аналогичной, работающей в той же отрасли. Да, требования соответствия или требования к информационной безопасности следует учитывать, однако, это больше вопрос подхода и технологии, а не необходимости работать исключительно общепринятым способом.
- Минимальное применение виртуализации и облачных вычислений, либо отказ от этих технологий вовсе, равно как использование сильно устаревших языков программирования. Аргументы, приведённые в первой части книги (в частности, инфраструктура как программный код, автоматизированное управление конфигурациями) показывают необходимость использования облачных вычислений. Компании, ограниченно использующие виртуализацию, будут иметь известные затруднения в организации DevOps. Однако, выбор тех или иных технологий — решение конкретной компании, и если новые управленческие инструменты требуют применения новых информационных технологий, то соответствующие изменения могут быть запланированы и воплощены в жизнь.

Очевидно, что наличие одного из ограничивающих факторов не делает DevOps невозможным в данной компании. Некоторую пользу можно получить и в сложных условиях, а многие ограничения можно обойти тем или иным способом. Также очевидно, что совокупность ограничивающих факторов усложняет применение DevOps всё больше и больше. Когда наступает (и наступает ли) тот предел, при котором ограничения складываются в непреодолимый барьер — неизвестно.

Опасность культа карго

Огромное количество команд, стремящихся освоить новые управленческие инструменты, не придают должного внимания слову «управленческие», фокусируясь на практиках. Сейчас модно строить разработку итеративно? Хорошо, мы организуем у себя двухнедельные спринты. Все вокруг устраивают ежедневные

Scrum-встречи? Отлично, у нас такие теперь есть. Говорят, визуализация в виде канбан-досок имеет смысл? Прекрасно, мы заведём у себя канбан. Конвейер DevOps без автоматизации не бывает? Что ж, поручим ребятам выбрать и настроить несколько систем. И так далее.

Данное поведение, при котором вместо целей, сути и принципов акцент смещается в сторону ритуалов, имеет название **культа карго** (англ. Cargo — товар). Понятие было впервые применено в 1945 году в области, не имеющей никакого отношения к информационным технологиям — антропологии. Учёные, изучавшие обычаи и особенности Папуа-Новой Гвинеи, выявили и обобщили явление, при котором, по мнению аборигенов, наличие материальных и духовных благ зависит в большей степени от воли духов и богов. Для получения таких благ необходимо совершать определённые действия и обряды, как правило — под руководством шамана или старейшины. Примеры и подтверждения культа карго были обнаружены и в более ранние време-на — самым давним документированным примером является культ на островах Фиджи в 1885 году. Говорят, некоторые проявления такого культа сохранились в отдельных частях Океании до наших времён.

Бездумное воспроизведение ритуалов гибкой разработки программного обеспечения, бережливых практик или DevOps-затей в надежде ускорить вывод продуктов на рынок встречается в практике различных компаний чаще, чем следует.

Заключение

DevOps имеет свои истоки, предпосылки появления. Для возникновения DevOps-движения к 2010-м годам сложились определённые условия, сформировавшие как потребность, так и возможность строить разработку и эксплуатацию информационных технологий иначе.

DevOps — не лекарство от всех болезней, как его зачастую преподносят различные «евангелисты». С его помощью можно решать три актуальные и непростые задачи: уменьшать время вывода на рынок, снижать технический долг и устранять хрупкость информационных систем.

DevOps опирается на мощный фундамент, основанный на бережливом производстве и гибкой разработке программного обеспечения. Некорректно утверждать, что DevOps — лишь использование уже известных идей; напротив, DevOps не только расширяет упомянутый фундамент, но и привносит несколько важных новых принципов.

Основываясь на этих принципах, можно искать, придумывать и применять практики. Многие из них будут непривычны для ИТ-отделов, работающих традиционным образом, однако за каждой из практик стоит достаточное основание, а зачастую холодный, в чём-то циничный расчёт.

Ещё один-два года назад можно было бы спорить о том, что такое DevOps, что входит в это понятие, что находится за границей, зачем всё это нужно и из чего состоит. Однако, к 2018 году в этих вопросах картина стала предельно ясной. Новые технологичные компании, созданные в последние пять лет, уже не представляют себе работу иной, для них DevOps — естественная часть корпоративной культуры, даже если само слово не произносится ежеминутно и не размещено на флаге. Традиционные компании с унаследованными ИТ-инфраструктурой, ИТ-решениями, процессами и персоналом ограничены в гибкости, однако активно присматриваются к новой модной теме, делают первые шаги, экспериментируют, ошибаются, учатся. Некоторые из них демонстрируют ошеломительные достижения, другие строят планы и питают надежды. Наибольшее число открытых вопросов, требующих поиска ответов, связано именно с корпоративными информационными технологиями. Если с техническими вопросами (например, как организовать конвейер развёртывания) всё более-менее понятно, то самый интересный вопрос— как получить управленческую пользу от DevOps в традиционных компаниях.