Олег Скрынник

DevOps для ИТ-менеджеров

Концентрированное структурированное изложение передовых идей

Второе издание

ДМК Пресс
издательство
Москва, 2019

УДК 004.45
ББК 65/290
С45

Скрынник О. В.
С45 DevOps для ИТ-менеджеров: концентрированное структурированное изложение передовых идей. – М.: ДМК Пресс, 2019. – 126 с.: ил.

ISBN 978-5-97060-692-6

В книге четко и последовательно изложены ключевые понятия, принципы и практики DevOps, дано определение DevOps и его место относительно других методологий управления ИТ, включая Agile. Эта книга не про автоматизацию, она акцентирована на разъяснение сути DevOps, построение команды и управление ИТ, без привязки к конкретным технологиям и инструментам. Книга не носит развлекательный характер, не является пособием «как делать DevOps», скорее, это настольный справочник для тех, кто хочет применять DevOps вдумчиво и со смыслом, со знанием дела оперируя понятиями и терминами. Благодаря тому что число технических терминов в книге сведено к минимуму, она отлично подойдет для чтения как специалистам в области информационных технологий, так и руководителям бизнес-подразделений и владельцам компаний.

УДК 004.45
ББК 65/290

ISBN 978-5-97060-692-6

Содержание

Об авторе

Олег Скрынник – управляющий партнер компании Cleverics.

В области информационных технологий работает более 20 лет, в том числе на руководящих должностях более 15 лет. Имеет опыт построения и реорганизации работы подразделений ИТ нескольких крупных компаний (предприятия сферы услуг, финансовые учреждения, промышленные предприятия). Постоянно использует этот опыт и знания при выполнении проектов, а также с удовольствием делится ими со слушателями курсов, мастер-классов и деловых игр.

Соучредитель некоммерческого партнерства «Форум по ИТ-сервис-менеджменту» (itSMF Russia). Имеет публикации в профильных СМИ, регулярно выступает на конференциях и семинарах. Ведет свой блог на портале Real ITSM (www.realitsm.ru).

Победитель конкурса статей «ITSM в России – 2014» (1-е место).

Победитель конкурса статей «ITSM в России – 2017» (1-е место).

Формальная сертификация:

- EXIN DevOps Master;
- ITIL® Expert;
- IT Service Manager;
- Microsoft® Certified Systems Engineer;
- Accredited GamingWorks™ Trainer.

Благодарности

Я очень благодарен своей семье и друзьям. Не стану утверждать, что они сильно помогли в написании книги – к счастью, мои близкие не связаны с такими вещами, как DevOps. Однако они совершенно точно пострадали: в некоторые моменты с июля по декабрь 2017 года я вдруг становился задумчивым и не реагировал на внешние раздражители, а иногда требовал соблюдения тишины по вечерам.

Я также благодарен своим коллегам по Cleverics. Организация собственного дела совместно с лучшими известными мне умами – безусловно, одно из самых значимых решений моей жизни. Чувство единства в целях и принципах, свободы в принятии решений, ответственности за результаты, ощущение плеча коллег ровно в ту минуту, когда поддержка наиболее необходима, – все это позволило выкроить немного времени для структурирования и фиксации мыслей о DevOps и превращения их в полноценную книгу.

Наконец, ничуть не менее я благодарен своим клиентам, которые не устают удивлять интересными задачами, радовать новыми вызовами, бесконечно загружать курсами, играми и семинарами, требовать лучшего и большего, буквально не давая возможности стоять на месте.

Предисловие автора

Эта книга написана ИТ-менеджером для ИТ-менеджеров. Она показывает DevOps не как явление, связанное с новыми инструментами автоматизации, приемами программирования или технологиями. Она объясняет управленческие аспекты DevOps для тех, кто профессионально занят *управлением* информационными технологиями.

Она отличается от других книг структурностью изложения (возможно – чрезмерной) и попыткой полностью охватить такое явление, как DevOps, на базовом, фундаментальном уровне. Это не означает поверхностного изложения, достаточного для первого знакомства с новой предметной областью. Под фундаментальностью подразумевается именно построение фундамента, основы – я рассказываю об истоках DevOps, неизбежности появления, ключевых предпосылках и их отражении в практиках, про сами практики и принципы, на которых они основываются.

Несмотря на обилие литературы по данной теме, это та самая книга, которой мне очень не хватало, когда я сам изучал DevOps. Я вижу своей задачей максимально четкое, структурное и лаконичное рассмотрение непростой, но очень интересной темы. Смею надеяться, что в данной книге нет ни одного лишнего слова, и напротив – есть все необходимые слова.

Москва, 2019

Глава 1

Что такое DevOps

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 придет время следующих, переосмысленных способов и техник, опирающихся на новые знания, опыт и технологии. Большую часть времени методы управления развиваются эволюционно, путем 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 вперед в вопросах эффективности и рациональности применения информационных технологий.

В качестве наглядного примера можно привести переход в управлении ИТ-деятельностью от принципа управления ИТ-системами к управлению ИТ-услугами. Начавшись около 20 лет назад, это изменение взгляда на менеджмент дало возможность первопроходцам получить значительные конкурентные преимущества. Появляющиеся новые практики после апробации становились так называемыми лучшими практиками, используемыми лидерами. Часть лучших практик постепенно переходила в статус общепринятых норм, некоторые из которых принимали вид отраслевых стандартов (рис. 1.1). Разумеется, определенная часть организаций не использовала в своей работе ни лучшие практики, ни стандарты – пока еще не все сферы экономической деятельности испытывают существенную зависимость от информационных технологий.

Рассмотрим, к примеру, управление ИТ-услугами. В 80–90-е годы XX века возникла идея предоставления ценности от применения информационных технологий в виде услуг и организации ИТ-деятельности в форме процессов. Отдельные европейские компании стали первопроходцами, разрабатывая новые практики организации работы и подходы к решению управленческих задач. Часть таких практик, например выделение функции Service Desk, отделение инцидентов от проблем, управляемая и контролируемая обработка изменений в ИТ-инфраструктуре и др., в 2000–2001 годах была сформулирована в ключевых публикациях, таких как библиотека ITIL[1]. Это позволило им

[1] https://www.axelos.com/best-practice-solutions/itil.

перейти в разряд лучших практик и начать использоваться не только лидерами, но и «догоняющими» организациями. Со временем, в 2002 году, появился первый стандарт BS 15000–1:2002 «Управление ИТ-услугами», задавший определенную норму, которой должны следовать те, кто стремится к построению целостной системы управления ИТ-услугами. При этом практики, публикации и стандарты не останавливаются в своем развитии (рис. 1.2).

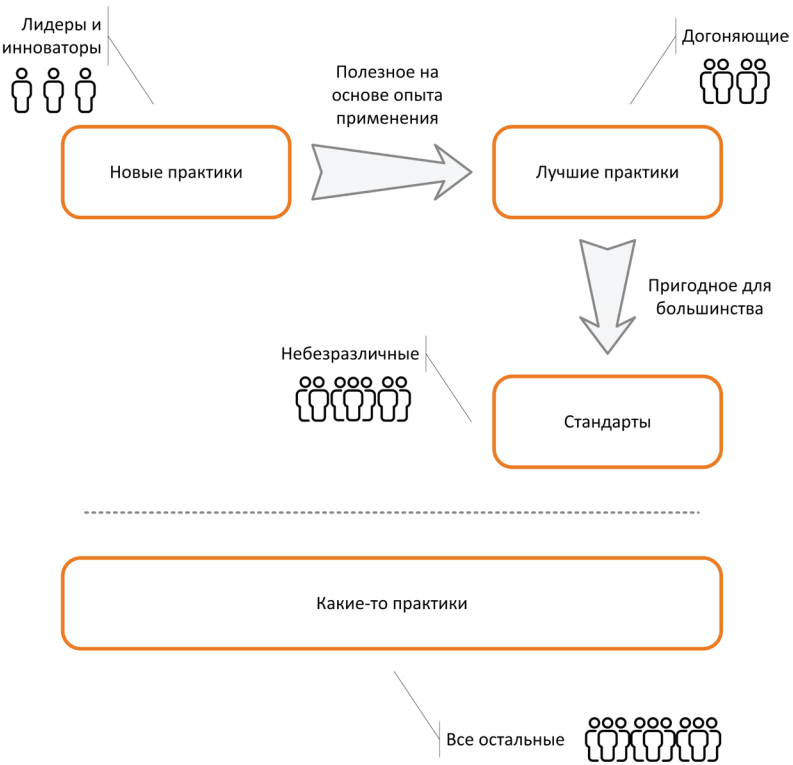

Рис. 1.1 ❖ Появление и использование новых практик

Аналогичная история происходит в настоящее время в развитии гибких подходов к разработке программного обеспечения. Однако назревающая здесь революция затрагивает более значительную область, чем только разработка ПО, и по масштабу последствий, возможно, встанет в один ряд с ITSM (IT Service Management, управление ИТ-услугами).

Новые, появляющиеся практики получили этикетку «DevOps»[1] – настолько же далекую от вкладываемого смысла, насколько ITIL далек от понятия «библиотека», а COBIT – от целей контроля.

[1] Сокр. от англ. Development & Operations.

Рис. 1.2 ❖ Развитие практик

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[1].

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.

Тем не менее само явление весьма достойно изучения. Для полного понимания сути DevOps необходимо рассмотреть предпосылки к появлению как идеи, так и связанного с ней движения.

Истоки

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.

[1] http://www.isaca.org/COBIT/Pages/FAQs.aspx.

Развитие гибких методов разработки программного обеспечения

В конце прошлого столетия доминирующей методологией разработки программного обеспечения была так называемая «водопадная модель» (рис. 1.3): последовательное выполнение заранее определенных 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. Дополнительным отличительным признаком такой модели является функциональная специализация исполнителей отдельных этапов: аналитиков, архитекторов, разработчиков, тестировщиков и т. д.

Рис. 1.3 ❖ Пример водопадной модели разработки программного обеспечения

При разработке крупных информационных систем с конечной функциональностью, которую можно 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 веб-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, 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. 1.4).

Рис. 1.4 ❖ 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. Кен Швейбер (Ken Schwaber) выпустил несколько публикаций о Scrum[1]. Кент Бек (Kent Beck) опубликовал книгу об экстремальном программировании, XP[2]. 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, притом что задача ставилась намного более широкая. Требовалось что-то, что позволит упростить и ускорить весь жизненный цикл программного обеспечения.

[1] Например, *Schwaber K.* Agile Software Development with Scrum. 2001. ISBN: 978-0130676344.

[2] *Beck K.* Extreme Programming Explained: Embrace Change. 1999. ASIN: B01FKT01PY.

В 2001 году К. Швейбер, К. Бек, а также еще 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. Один из авторов манифеста, Роберт Мартин (Robert C. Martin), поясняет[1]:

> *«При использовании правильных дисциплин и правильного минимального процесса может возникнуть и развиться доверие между разработчиками и бизнесом. Бизнес начнет доверять разработчикам, вместо того чтобы думать, что они ленивые, продажные, противные существа, а разработчики начнут обращать внимание на бизнес и осознают, что те являются разумными, рациональными существами, а не с какой-то другой планеты».*

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

Манифест Agile
(полный текст, цитируется по официальному переводу[2])

Люди и взаимодействие	важнее	процессов и инструментов
Работающий продукт	важнее	исчерпывающей документации
Сотрудничество с заказчиком	важнее	согласования условий контракта
Готовность к изменениям	важнее	следования первоначальному плану

То есть, не отрицая важности того, что справа, мы все-таки больше ценим то, что слева. Мы следуем таким принципам.

1. Наивысшим приоритетом для нас является удовлетворение потребностей заказчика благодаря регулярной и ранней поставке ценного программного обеспечения.
2. Изменение требований приветствуется, даже на поздних стадиях разработки. Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.
3. Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.
4. На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.
5. Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.
6. Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.
7. Работающий продукт – основной показатель прогресса.

[1] https://www.youtube.com/watch?v=hG4LH6P8Syk, а также https://www.aaron-gray.com/a-criticism-of-scrum/.

[2] http://agilemanifesto.org/iso/ru/manifesto.html.

8. Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.
9. Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
10. Простота – искусство минимизации лишней работы – крайне необходима.
11. Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
12. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

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, выдает 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 дает 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 собственно разработки имеется еще 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 жесткой, основанной на дорогом аппаратном обеспечении, которое было приобретено достаточно давно, бюджеты на закупку и настройку выделялись непросто, да и собственно 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.

> Для борьбы с хрупкостью некоторые организации применяют формализованный и автоматизированный процесс управления изменениями, призванный структурировать поток изменений и минимизировать риски при их выполнении.

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[1]. 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 распространенных в последующем машин на архитектуре Intel x86 (80–90-е годы)[2]. Рисунок 1.5 показывает количество ключевых событий, связанных с виртуализацией, на отрезке с 1964 по 2008 год (график не случайно завершается данным годом, что станет видно из дальнейшего изложения).

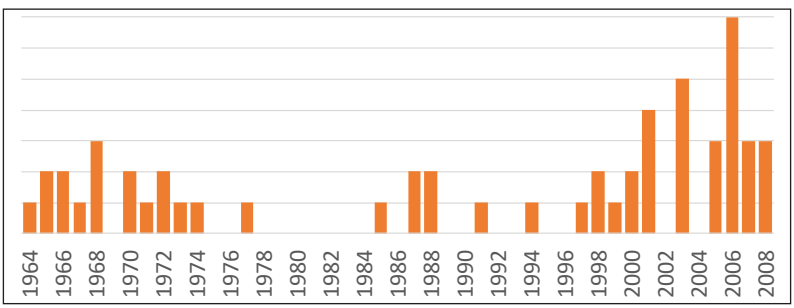

Рис. 1.5 ❖ Распределение во времени ключевых событий, связанных с виртуализацией

[1] https://en.wikipedia.org/wiki/IBM_CP-40

[2] Интересно отметить, что по утверждению Д. Хамбла в те годы случился некоторый период времени, когда компания IBM избегала рекомендовать продукты виртуализации своим клиентам, так как это влияло на продажи оборудования.

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 еще быстрее. До середины 90-х годов прошлого века телекоммуникационные компании предлагали своим клиентам организацию частных глобальных вычислительных сетей (WAN, Wide Area Network) путем 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 идет «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 удаленными 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 свое 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, надежными 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*».

Национальный институт стандартов и технологий США определил пять ключевых характеристик облачных вычислений[1]:

1) самообслуживание по требованию – потребитель самостоятельно определяет и изменяет вычислительные потребности, такие как серверное время, скоро-

[1] http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

сти доступа и обработки данных, объем хранимых данных без взаимодействия с представителем поставщика услуг;
2) универсальный доступ через сеть – услуги доступны потребителям по сети передачи данных через стандартные механизмы, которые способствуют использованию гетерогенными тонкими или полными клиентскими платформами;
3) объединение ресурсов – поставщик услуг объединяет ресурсы для обслуживания большого числа потребителей в единый пул для динамического перераспределения мощностей между потребителями в условиях постоянного изменения спроса на мощности;
4) эластичность – услуги могут быть предоставлены, расширены, сужены в любой момент времени, без дополнительных издержек на взаимодействие с поставщиком, как правило, в автоматическом режиме, соизмеримо со спросом;
5) учет потребления – поставщик услуг автоматически измеряет потребленные ресурсы на определенном уровне абстракции и на основе этих данных оценивает объем предоставленных потребителям услуг.

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 счет облачного провайдера в конце месяца.

То есть необходимая инфраструктура создается с помощью программного кода. И не только создается, но и может управляться как программный код – с хранением версий, отслеживанием изменений, отладкой, повторным использованием прошлых наработок и т. д. Более подробно данные аспекты будут рассмотрены в главе 3 «Принципы».

В завершение отметим также вторую жизнь, которую получили давно придуманные технологии. К примеру, виртуализация на уровне операционной системы была доступна во многих UNIX-системах еще в 80-е годы прошлого столетия. Однако серьезный коммерческий успех этой технологии, которую чаще стали называть контейнеризацией, пришел 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

> «Когда вы слышите, что кто-то говорит о неизбежности, – скорее всего, за этим скрываются коммерческие компании, очень старающиеся сказку сделать былью».
>
> *Ричард Столман (Richard Stallman), основатель Free Software Foundation и создатель операционной системы GNU, об облачных вычислениях*[1]*, 2008*

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.

Трезво воспринимая слова Р. Столмана, приведенные выше (а с облачными вычислениями он, похоже, сильно ошибся), можно предполагать, что появление чего-то, аналогичного DevOps, было лишь вопросом времени.

Определение

Только очень самоуверенные либо бесконечно некомпетентные люди, а также общепризнанные гуру могут всерьез рассуждать о каком-либо явлении, не дав ему предварительно определения либо не опираясь на общепринятое определение. С DevOps, к сожалению, ситуация отнюдь не проста.

Некоторые эксперты пытаются придумать что-то свое, близкое их пониманию. Другие утверждают, что определить DevOps в настоящий момент невозможно, так как это, скорее, явление, движение, идея, но не дисциплина и не

[1] https://www.theguardian.com/technology/2008/sep/29/cloud.computing.richard.stallman.

методология. Третьи сообщают, что DevOps у каждого свой, и приводят известную аналогию про нескольких слепых, ощупывающих слона: один говорит, что, скорее всего, это дерево, другой – что коврик, третий – что змея, и т. д.

За то время, которое я занимаюсь данной темой, мне удалось прочесть большое количество книг и интернет-публикаций, пообщаться с совершенно разными людьми, вовлеченными в DevOps-движение – как в России, так и в Европе, посетить тематические учебные курсы и сдать несколько международных профильных экзаменов. На мой взгляд, опасения по поводу невозможности определения DevOps несколько преувеличены. Разумеется, сколько людей – столько и мнений, а в случае с консультантами все еще серьезнее: два консультанта – минимум три мнения. Однако наличие системного склада ума, высшего технического образования и опыта консультирования в сфере ИТ-менеджмента дает возможность подойти к вопросу четко и структурно. Не претендуя на универсальность или истину в последней инстанции, я составил следующее определение:

> *DevOps – продолжение идей гибкой разработки программного обеспечения и бережливого производства, примененное к полной цепочке создания ценности в ИТ, позволяющее добиваться большего от современных информационных технологий за счет культурных, организационных и инструментальных изменений.*

В данном определении есть важные моменты, которые необходимо подчеркнуть.

Во-первых, представляется важным указать на то, что DevOps не подменяет собой гибкие и бережливые практики, но как бы вбирает их в себя. Общение с коллегами, клиентами, слушателями учебных курсов показывает, что те, кто незнаком с идеями гибкой разработки, открывают для себя в DevOps очень много нового и любопытного. Те же, кто имеет соответствующую подготовку и опыт, удивляются большому количеству пересечений между практиками DevOps и практиками, скажем, Lean, Scrum и Kanban. На мой взгляд, не совсем корректно называть такое явление «пересечением». Скорее, речь про заимствование, расширение идей гибкой разработки и бережливого производства. Более подробно данный вопрос рассматривается в главе 2 «Фундамент».

Во-вторых, сама суть DevOps заключается в том, что ИТ-подразделение вместе с бизнесом думает не только про разработку программного обеспечения, но про всю цепочку создания ценности. Эта цепочка начинается с генерации новых идей совместно с представителями бизнес-подразделений, проходит через разработку, тестирование, развертывание, вплоть до эксплуатации. Такой взгляд дает возможность системного анализа, поиска и устранения узких мест во всей цепочке, налаживания механизмов предоставления обратной связи не только из конца цепочки в ее начало, но и внутри, между шагами, равно как и в пределах одного шага. Перечисленным следствиям из данного определения – системному взгляду, работе с ограничениями и организации обратной

связи – в DevOps уделяется максимальное внимание, о чем будет подробно рассказано позже.

В-третьих, следует сделать акцент на ожидаемой пользе от применения DevOps, которая заключается в большей отдаче от информационных технологий. Согласно классическому представлению, информационные технологии позволяют организациям получать больше выгод (через создание новых возможностей или устранение имеющихся ограничений), снижать риски и оптимизировать ресурсы. Правильно построенный DevOps позволяет учесть все три перечисленных аспекта. Некорректно было бы утверждать, что организации не могут получать пользу от информационных технологий традиционными способами, без применения DevOps. Однако DevOps позволяет получить *больше* пользы, которая может выражаться в ускорении вывода на рынок новых и модифицированных продуктов, быстрой реакции на потребности клиентов, увеличении доступности и устойчивости ИТ-систем, более эффективном использовании ограниченных ресурсов. Несколько подробнее данная тема будет изложена в разделе «Зачем нужен DevOps».

И наконец, в завершающей части определения есть явное указание на три ключевые составляющие: культурные, организационные и инструментальные средства (рис. 1.6). По сути, это та самая старая мантра про процессы, людей и технологии. Опыт первопроходцев DevOps, а также их последователей показывает, что ее важность все так же сильна.

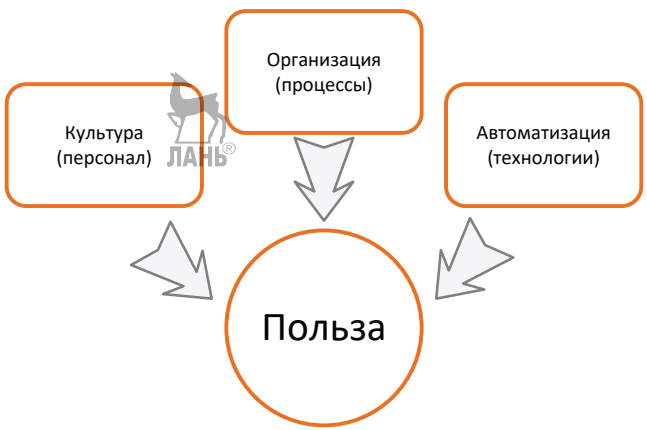

Рис. 1.6 ❖ Три ключевых компонента

Приведем данное выше определение с разбивкой на существенные составные части, позволяющей еще раз сделать необходимые акценты.

DevOps:

а) *продолжение идей гибкой разработки программного обеспечения и бережливого производства*;

б) *примененное к полной цепочке создания ценности в ИТ*;
в) *позволяющее добиваться большего от современных информационных технологий*;
г) *за счет культурных, организационных и инструментальных изменений.*

Зачем нужен DevOps

«Послушайте! Ведь если звезды зажигают – значит, это кому-нибудь нужно?»

В. Маяковский[1]*, 1914*

Некоторые подходы к управлению появляются как отражение фантазии автора (возможно, эксперта, или даже гуру) – этакие теоретические изыскания. Их применимость либо неприменимость доказывается адептами и последователями, пытающимися использовать новые приемы в своей работе и в организации работы других сотрудников.

Иные подходы рождаются как ответ на вполне насущные потребности. Они создаются не по заказу Британской короны и не группами специально привлеченных консультантов, но практиками, ищущими возможности устранить какие-либо сложности или ограничения, либо более эффективно использовать имеющиеся ограниченные ресурсы, либо создать новые бизнесы, новые ниши, новые инструменты решения управленческих задач.

Похоже, что DevOps ближе ко второй группе, нежели к первой. Подробности появления будут рассмотрены в разделе «История возникновения»; пока же сконцентрируемся на основных проблемах, которые различные организации пытаются решить с помощью DevOps.

Уменьшение времени 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 расчет (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;

[1] https://ru.wikisource.org/wiki/%D0%9F%D0%BE%D1%81%D0%BB%D1%83%D1%88%D0%B0%D0%B9%D1%82%D0%B5!_(%D0%9C%D0%B0%D1%8F%D0%BA%D0%BE%D0%B2%D1%81%D0%BA%D0%B8%D0%B9).

- 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, ее 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, притом что бизнесу хотелось бы сократить ее до месяцев. Бизнес-обоснование здесь прозрачно: за время разработки нового продукта рынок может измениться настолько, что собственно 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, дает 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 все возрастающую скорость изменений. Одна из наиболее наглядных иллюстраций данного тезиса – закон ускоряющихся возвратов (Law of Accelerating Returns), сформулированный в 1999 году Реем Курцвайлом[1] (Ray Kurzweil). 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 все 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…

> «Авторам сценариев для кинематографа приходится туго. За меняющейся с быстротой молнии обстановкой положительно не угнаться. …Пока успеешь написать сценарий, пока его разыграют, пустят по прокатным конторам и пока он дойдет до экрана, смотришь: выпархивает что-либо более новое, более отвечающее моменту…
>
> В точно таком же положении оказываются авторы… разных и злободневных пьесок. Им придется скоро творить прямо за утренним кофе, за чтением газетных телеграмм. С тем чтобы к полудню пьеса была готова для генеральной репетиции, а вечером появлялась перед публикой. Только при этом непременном условии можно даже уловить момент».
>
> «Театр», московская ежедневная театральная газета[2], август 1917

[1] http://www.kurzweilai.net/the-law-of-accelerating-returns.

[2] https://project1917.ru/groups/teatr.

Вторая сложность описанного выше процесса заключается в необходимости четкой 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 еще больше увеличивает фактическое время вывода на рынок и требует еще 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. Скрупулезному и детальному планированию они предпочитают эксперименты, а слово «идея» заменяют на «гипотезу». В этом случае процесс выглядит примерно так:

- рождение гипотезы, разработка методов оценки ее 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. 1.7). Один раз войдя в такую реку, нащупав новую нишу и новые возможности, компании будет необходимо всегда искать изменяющееся русло. Притом, 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 приведенного 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. Они будут более подробно рассмотрены в главе 4 «Основные практики». Сейчас же важно сделать следующее замечание: наивно надеяться, что применение техник DevOps для

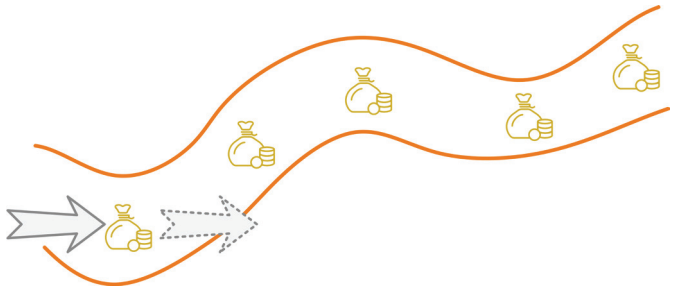

Рис. 1.7 ❖ Река с деньгами

ускорения работы ИТ-отдела одновременно приведет к сокращению затрат на ИТ. Скорее наоборот, расходы на информационные технологии вырастут, что обусловлено в первую очередь увеличением количества 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 позволяет равномерно распределять между ними нагрузку (рис. 1.8).

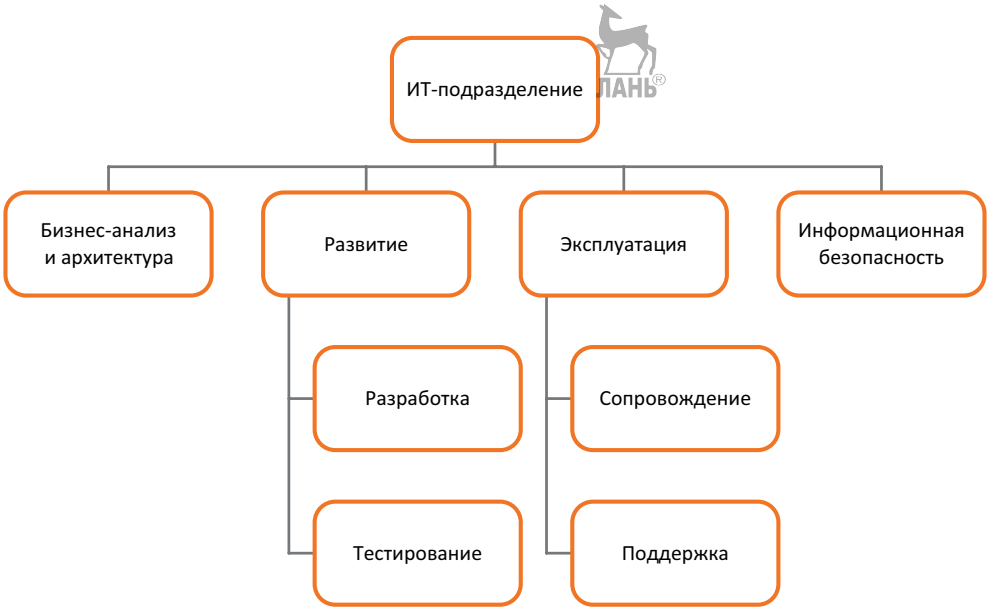

Рис. 1.8 ❖ Функциональная структура традиционного ИТ-подразделения

В отличие от такой схемы, в DevOps деление специалистов производится по командам, и каждая команда работает над своим продуктом (рис. 1.9). Будучи самодостаточной, команда включает в себя и владельца продукта, и архитекторов, и разработчиков, и тестировщиков, и ответственных за эксплуатацию, и за информационную безопасность. При большом количестве команд, каждая из которых сфокусирована исключительно на своем продукте, равномерность загрузки специалистов обеспечить сложнее, что может приводить к неполной утилизации персонала, а значит – к повышению расходов на него (эта тема будет продолжена в разделе «Необычные команды»).

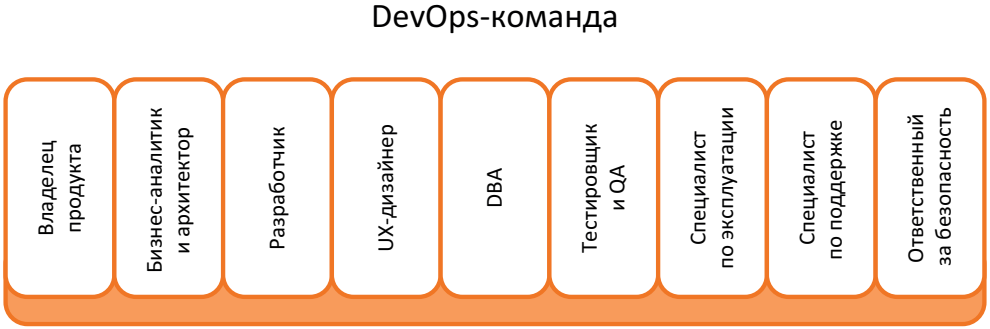

Рис. 1.9 ❖ Пример состава DevOps-команды

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-партнере, а не центре затрат.

Снижение технического долга

Понятие технического долга предложил Уорд Каннингем (Ward Cunningham) в 1992 году[1]. 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 пробле-

[1] http://wiki.c2.com/?WardExplainsDebtMetaphor.

ма 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[1] (рис. 1.10).

	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»*

Рис. 1.10 ❖ Классификация технического долга по М. Фаулеру

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.

[1] https://martinfowler.com/bliki/TechnicalDebtQuadrant.html.

Более того, можно утверждать, что некоторые общеизвестные практики, предлагаемые библиотекой ITIL, будучи примененными 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 том числе 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 в разделе «Развитие гибких методов разработки программного обеспечения», 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, притом 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 все это продолжало работать дальше;

- документирование программного кода, архитектуры, инфраструктуры, технических решений и даже контрактных обязательств оставляет желать лучшего.

Джин Ким (Gene Kim), Джез Хамбл (Jez Humble), Патрик Дебуа (Patrick Debois) и Джон Виллис (John Willis) отмечают[1], 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. Все 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 – путем ее 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»[2] Нассим Николас Талеб (Nassim Nicolas Taleb) рассуждает об особенностях сложных систем и вводит следующую классификацию: хрупкие системы, устойчивые системы и антихрупкие системы. Приведенное 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 учетом 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).

---

[1] *Kim G., Humble J., Debois P., Willis J.* The Devops Handbook: How To Create Worldclass Agility Reliabil-ity And Security In Technology Organizations. 2016. ISBN 978-1-942-78800-3. Раздел «Downward Spiral In Three Acts».

[2] *Taleb N.* Antifragile: Things That Gain from Disorder. 2012. ISBN 978-1400067824.

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 приведет к решению обозначенных задач, этого недостаточно. Необходимо самым серьезным образом менять *культуру работы* ИТ-организации, чтобы изменились не только применяемые инструменты, приемы и техники, но и *отношение ИТ-персонала* к ключевым вопросам работы компании: роли заказчика, ценности от 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 приведет к явлению, известному как культ карго (англ. Cargo cult). Более подробно данный тезис рассматривается в разделе «Культ карго».

Однако отвлечемся на минуту от важных вопросов и вспомним историю возникновения DevOps.

История возникновения

Казалось бы, прагматичное рассмотрение какой-либо темы не требует погружения в историю вопроса – какая в целом разница, кто, когда, с кем собрался, что обсудил и что придумал?

Если подходить к DevOps как к набору техник, которые лишь нужно «внедрить» в своей организации, то действительно – данный раздел можно и пропустить. Однако, судя по всему, успех DevOps-трансформации любой компании напрямую зависит от персонала. Именно он должен изменить текущую практику своей работы. Именно он может этого не сделать, не разделяя новых ценностей, не понимая ответов на вопросы «зачем?» и «почему именно сейчас?».

Поэтому потратим немного времени на краткий экскурс в историю, так как за всеми излагаемыми ниже событиями стоят конкретные персоналии, имевшие и имеющие свою мотивацию.

Как было рассмотрено в разделе «Развитие гибких методов разработки программного обеспечения», ко второй половине 2000-х годов различные методики вроде Agile, Scrum, XP и подобные им стали довольно активно применяться в области разработки ПО. Расширение их на зону ответственности эксплуатации, скорее всего, было лишь вопросом времени. Одна из первых документированных попыток была сделана в 2006 году, когда Марсель Вигерманн (Marcel Wegermann) опубликовал статью о применении принципов гибкой разработки к работе системных администраторов[1]: он предложил включать отдельные системные каталоги в систему контроля версий, использовать парную работу системных администраторов, проводить ретроспективы, относящиеся к эксплуатации.

В 2008 году на очередной конференции об Agile, проводившейся в Торонто, случилось сразу два знаменательных события. Эндрю Шейфер (Andrew Shafer) предложил включить в программу новый трек «Гибкая инфраструктура», а Патрик Дебуа сделал доклад «Гибкие эксплуатация и инфраструктура: насколько вы infra-gile?[2]». Их интересы во многом совпали, так как Э. Шейфер как раз переходил из разработки в эксплуатацию, а П. Дебуа в то время работал одновременно с разработчиками и поддержкой, наблюдая совершенно различные подходы к организации труда в каждой из команд – с 2007 года он занимался большим проектом миграции центра обработки данных. По его воспоминаниям, доклад не произвел большого впечатления на аудиторию. Другие участники той конференции рассказывают, что и аудитория-то собралась скромная. Однако дальнейшие события показали, что те, кто пришли, были очень заинтересованы в изложенных идеях, и сами идеи получили свое продолжение, притом довольно скорое.

В том же 2008 году Люк Канис (Luke Kanies), основатель компании Puppet Labs, выступал на конференции, посвященной открытому программному обеспечению, с докладом про управление конфигурациями, которое, по его мнению, нужно было выстраивать совершенно иначе, чем принято. Докладом заинтересовался Джон Виллис (John Willis), впоследствии оказавший сильное влияние на развитие идей DevOps. Стоит заметить, что самого термина DevOps в то время еще не было.

Слово DevOps было подхвачено сообществом после выступления Джона Олспо (John Allspaw) и Пола Хаммонда (Paul Hammond) на конференции Velocity в 2009 году. Доклад, называвшийся «10 развертываний в день»[3], произвел не-

---

1 *Davis J., Daniels K.* Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale. 2016. ISBN 9781491926437. Раздел «A History of Devops».

2 http://www.jedi.be/presentations/IEEE-Agile-Infrastructure.pdf.

3 http://velocityconf.com/velocity2009/public/schedule/detail/7641.

изгладимое впечатление на различные умы, так или иначе уже интересовавшиеся данной темой. П. Дебуа решил организовать первую специализированную конференцию DevOpsDays, которая прошла в городе Гент, Бельгия, в том же 2009 году[1]. Джин Ким, присутствовавший на докладе, в 2013 году выпустил книгу «Проект Феникс»[2] и основал компанию IT Revolution, занимающуюся популяризацией темы и организующую дважды в год мероприятия «DevOps Enterprise Summit[3]».

Появились и слово «DevOps», и сообщество энтузиастов…

Доклад «10 развертываний в день» в настоящее время считается отправной точкой движения DevOps.

DevOpsDays – наиболее популярные мероприятия в жизни DevOps-сообщества, охватывающие множество стран.

«DevOps Enterprise Summit» – самые крупные и наиболее представительные DevOps-конференции, как по количеству участников, так и по составу докладчиков.

Книга «Проект Феникс», похоже, является самой продаваемой книгой по DevOps в мире.

Компании «Puppet Labs» и «IT Revolution» входят в число наиболее заметных и влиятельных игроков на рынке DevOps.

Большинство из упомянутых выше персоналий считаются безусловными гуру в мире DevOps.

Оглядываясь назад, можно сделать следующие наблюдения. Во-первых, ключевые идеи DevOps возникли в результате интеллектуальной работы по поиску решений имеющихся управленческих задач. Во-вторых, у DevOps нет одного отца-основателя или группы товарищей – ключевые персоны не всегда были предварительно знакомы между собой, даже если мыслили в схожих направлениях. В-третьих, у DevOps нет и не может быть правообладателя, диктующего или определяющего развитие либо вводящего ограничения на использование (хотя отдельные жадные коммерческие компании уже резервируют за собой производные торговые марки, такие как «DevOps Foundation»). В-четвертых, тема DevOps настолько молода, что ожидать сборников рецептов или универсальных методов пока преждевременно.

Некоторые частые заблуждения

Главу 1 «Что такое DevOps» лучше всего завершить рассмотрением часто встречающихся заблуждений. Это поможет яснее очертить границы явления и позволит перейти к рассмотрению следующих, более специфичных вопросов.

---

1 https://legacy.devopsdays.org/events/2009-ghent/.

2 *Behr K., Spafford G., Kim G.* The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win. 2009. ISBN 978-0988262508.

3 https://events.itrevolution.com/.

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, путем 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 такой 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.

Для примера посмотрим на популярную модель SAFe (Scaled Agile Framework), предназначенную для помощи применения идей гибкой разработки к организациям среднего и крупного масштаба[1] (рис. 1.11).

Искомый DevOps находится в правой части модели, примерно посередине по вертикали, в разделе «Program». Судя по размеру шрифта, его значимость примерно соответствует значимости понятий «Backlog», «Kanban» и «Business Owners». Собственно, в описании SAFe сказано следующее[2]:

> «*Организации, использующие SAFe, внедряют DevOps для разрушения барьеров и создания условий [командам Agile] непрерывно предоставлять новые возможности своим конечным пользователям. С течением времени разделение между разработкой и эксплуатацией значительно сокращается. …Задача проста: доставлять ценность более часто*».

Это, безусловно, очень ограниченный взгляд на DevOps, минимум по трем 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 (см. главу 3 «Принципы»). Во-вторых, получение отдачи от DevOps требует более значительных культурных изменений в компании, чем это обычно происходит при применении Agile (см. раздел «Культ карго»). В-третьих, задачи, которые ставятся перед DevOps, не ограничиваются лишь ускорением поставки – есть также необходимость снижения технического долга и устранения хрупкости (см. раздел «Зачем нужен DevOps»).

---

[1] http://www.scaledagileframework.com/.

[2] http://www.scaledagileframework.com/devops/.

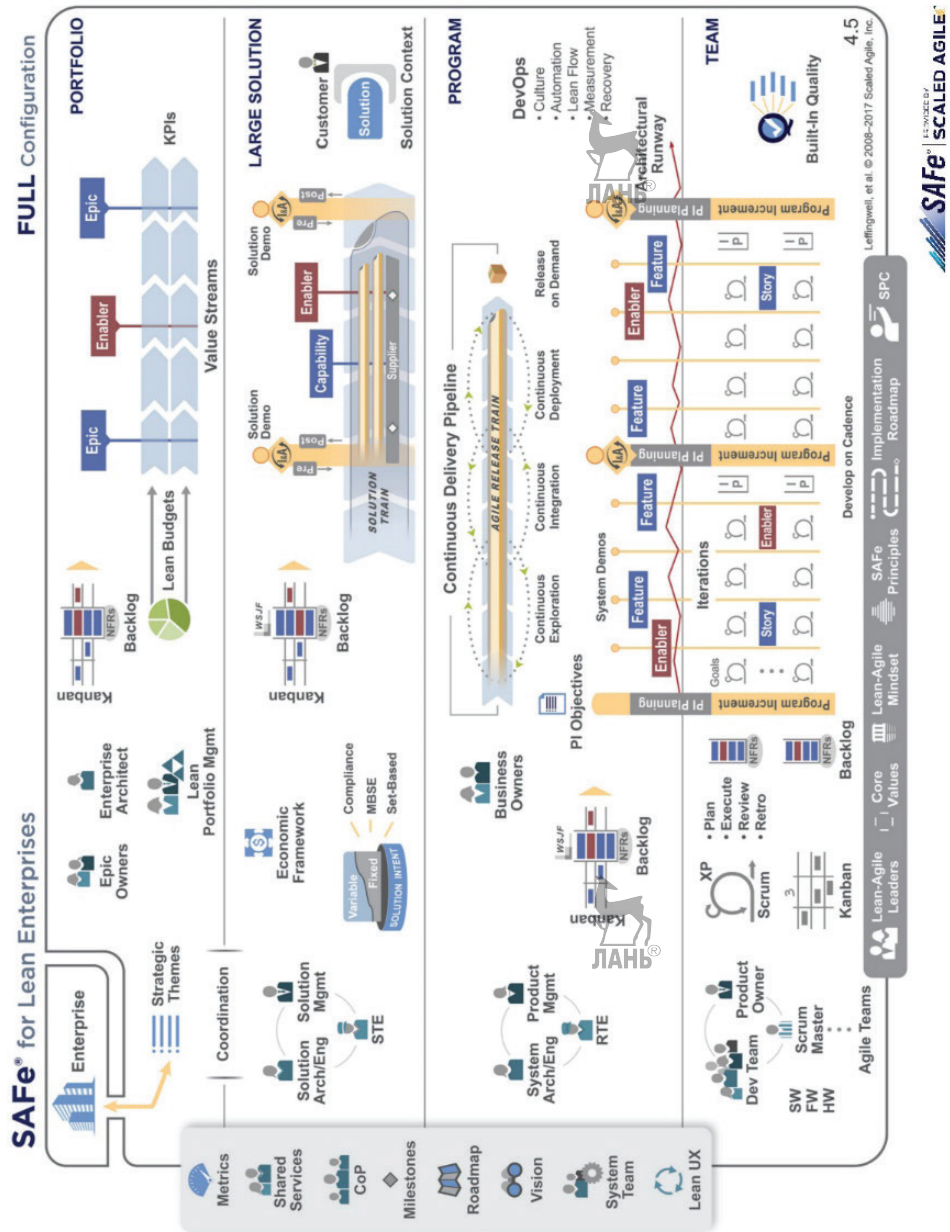

Рис. 1.11 ❖ Модель SAFe версии 4.5 (2017)

DevOps – это автоматизация и инструменты

> «Конференция о DevOps для разработчиков. Мировые эксперты. Технический хардкор без воды (18+)».
>
> *Рекламное объявление в поисковой системе Google, октябрь 2017*

Другая точка зрения сводится к слову «Автоматизация». Программных инструментов, помогающих работать современному ИТ-отделу, в последние годы развелось великое множество – они исчисляются сотнями (рис. 1.12). 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 вроде CA, HP и Microsoft, 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 всерьез обсуждают что-то за пределами ПО.

DevOps действительно зависит от наличия и работоспособности определенных 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. Все 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. Косвенным подтверждением независимости конкретной реализации DevOps от программного обеспечения является данная книга, в которой можно позволить себе достаточно детально изучить явление, не рассматривая ни одного программного продукта, а то и не упоминая их названий вовсе.

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.

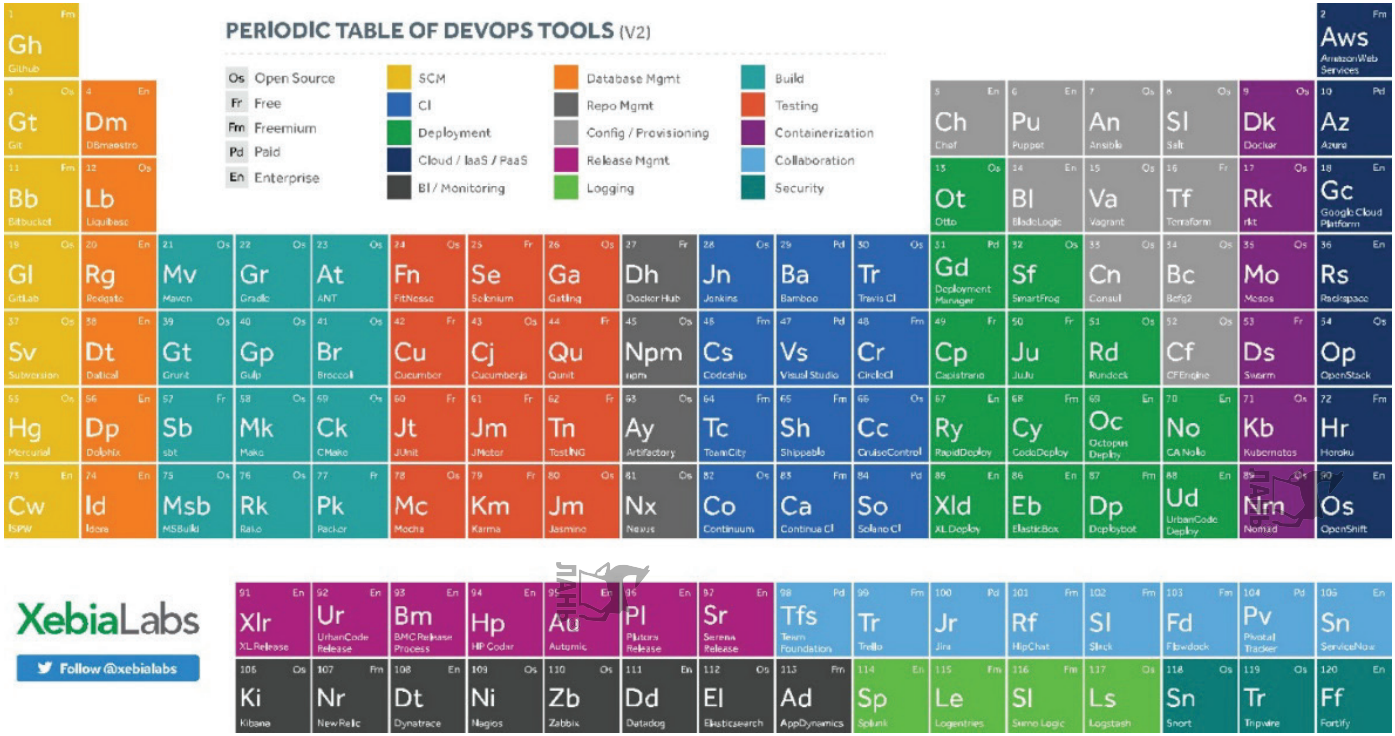

Рис. 1.12 ❖ «Периодическая таблица» инструментов DevOps версии 2 от компании Xebia Labs[1]

[1] https://xebialabs.com/periodic-table-of-devops-tools/.

«В продуктовую StartUp компанию для удаленного сотрудничества необходим DevOps/System Administrator (Bitrix)».

Объявление на сайте поиска сотрудников, октябрь 2017

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

Все это, разумеется, серьезные 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 в разделе «Уменьшение времени вывода на рынок».

Краткое резюме главы

Завершим данную главу книги коротким резюме, содержащим все основные изложенные сведения. Для упрощения понимания, в том числе при последующих возвратах к теме DevOps и общении с коллегами, будем использовать графическое отображение информации (рис. 1.13, 1.14).

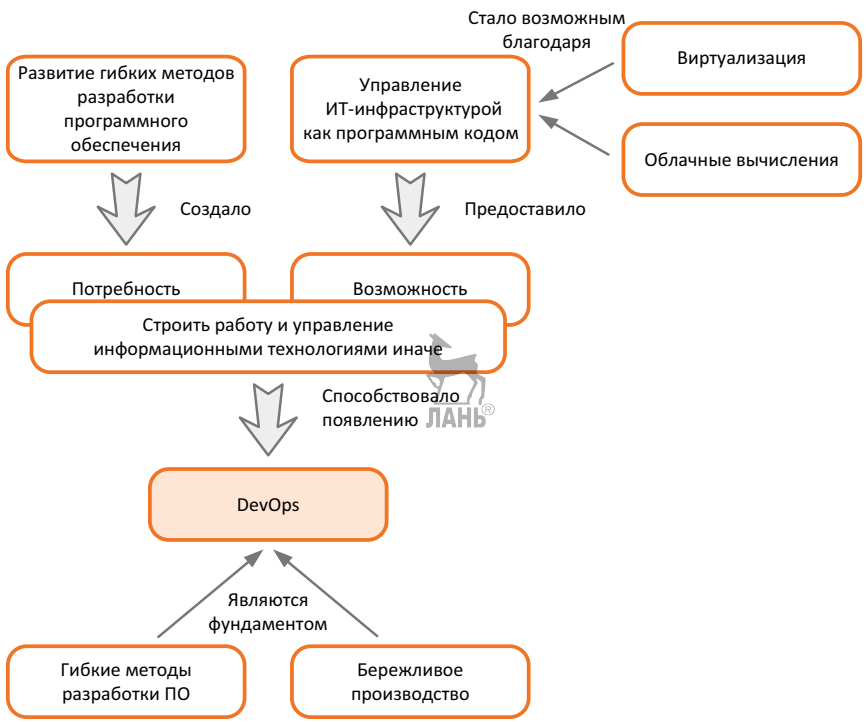

Рис. 1.13 ❖ Картина мира DevOps

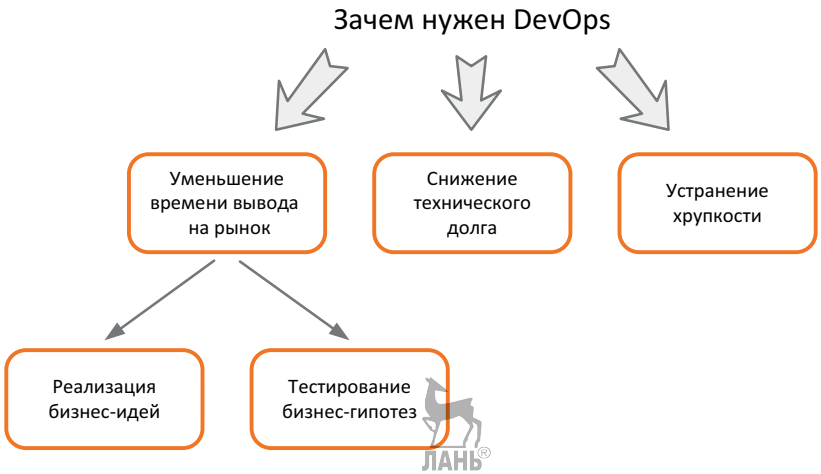

Рис. 1.14 ❖ Зачем нужен DevOps

Глава 2

Фундамент

Бережливое производство

Основные сведения

Как уже упоминалось в разделе «Определение», DevOps в существенной мере опирается на известные принципы и практики бережливого производства (Lean). Некоторые даже считают, что кроме них в DevOps, по большому счету, ничего больше нет, – такое утверждение не соответствует действительности. Для объяснения следует рассмотреть основные положения бережливого производства, чтобы хорошо представлять себе тот фундамент, на котором построен DevOps.

Очень упрощенно идею бережливого производства можно свести к выявлению и устранению потерь. Чтобы лучше представить себе суть данного утверждения, необходимо вспомнить про задачу, которую изначально пытались решить с помощью Lean.

В 1930–1940 годы небольшая компания «Toyota» увидела возможности в освоении рынка автомобилей. С одной стороны, платежеспособность населения в тот момент была низкой, а значит, продукция должна быть как можно более дешевой. С другой стороны, размер рынка был весьма скромным, а значит, применять принципы массового производства, позволяющие экономить на масштабе, не получилось бы. Компания решила найти свой путь, и дальнейшая история показала, что ей это удалось. Мозговым центром создания и развития новой технологии производства стал Таичи Оно (Taiichi Ohno), который представлял себе идеальную картину такой: производство начинается только после того, как клиент разместил заказ, при этом новый автомобиль должен быть поставлен клиенту немедленно. Получается, что для приближения к такой скорости производства необходимо как можно скорее выполнять только те операции, которые непосредственно направлены на создание продукта, в то же время устраняя все возможные потери.

Понятию «потери» в бережливом производстве уделяется много внимания: обычное, бытовое понимание дополняется и расширяется так, чтобы стать объектом управления на разных участках выполнения работы. На верхнем уровне потери разделяются на «мури», «мура» и «муда». *Мури* можно обозначить как работу сомнительной ценности, которую руководство возлагает на исполните-

лей ввиду неоптимально организованных процессов, привлечения с постоянной перегрузкой или повышенной сверхмеры интенсивностью. *Мура* обозначает уровень плавности, равномерности, предсказуемости потока выполнения работ, устранения разброса, флуктуаций. Под *мудой* подразумеваются потери, проявляющиеся в процессе выполнения работ, – места их проявления и свойства настолько неочевидны, что потребовалась дополнительная классификация таких потерь. Оригинальный список приведен ниже, аналоги потерь в ИТ взяты из публикаций Мэри и Тома Поппендик (Mary Poppendieck, Tom Poppendieck)[1].

Виды потерь на производстве и их аналоги в ИТ

Потери	**Пояснение в части ИТ**
Запасы Аналог потерь в ИТ: частично выполненная работа	Незавершенная работа не приносит ценности конечному клиенту, притом что ресурсы уже израсходованы. Реализацию ценности от незавершенной работы невозможно оценить ввиду отсутствия обратной связи от заказчика. Результаты частично выполненной работы (произведенной, например, для исключения простоя ресурсов) могут устареть либо не потребоваться в дальнейшем
Дополнительная обработка Аналог потерь в ИТ: дополнительные этапы	Под дополнительными этапами подразумеваются любые шаги любого процесса, за исключением анализа, программирования и развертывания приложения. В том числе документирование, согласование, планирование, составление отчетности и т. д.
Перепроизводство Аналог потерь в ИТ: дополнительные возможности	Любые дополнительные возможности расходуют ресурсы по всей цепочке создания ценности: на анализ, кодирование, тестирование, развертывание, эксплуатацию. При этом существенная часть функциональности типового программного обеспечения не используется клиентами, то есть не несет ценности. Кроме того, дополнительные возможности создают дополнительные потенциальные точки отказа
Транспортировка Аналог потерь в ИТ: многозадачность	Переключение между задачами приводит к потерям времени, связанным в том числе с концентрацией внимания и погружением в контекст. В общем случае суммарное время выполнения нескольких задач, если заниматься ими одновременно, будет существенно выше суммарного времени на выполнение тех же задач последовательно
Ожидание Аналог потерь в ИТ: ожидание	Ожидание чего-либо на любом участке замедляет всю цепочку создания ценности, что приводит к увеличению времени получения конечного результата. Типовые задержки, связанные с ожиданием в ИТ: ожидание принятия решения, ожидание назначения или освобождения ресурсов, ожидание завершения документирования предыдущих шагов, ожидание принятых в организации циклов (например, бюджетных или работы каких-либо комитетов)
Перемещение Аналог потерь в ИТ: передача работы и информации	Получение сведений, необходимых для выполнения работы, может требовать значительного времени или ресурсов. Передача артефактов, возникающих на данном участке работы, может производиться нелинейно, сложным процессом и также требовать затрат времени или ресурсов. Для всего этого зачастую требуется в том числе физическое перемещение персонала или документации
Дефекты Аналог потерь в ИТ: дефекты	Негативные последствия дефектов могут быть оценены как влияние данных дефектов на работоспособность ИТ-системы с учетом времени, на протяжении которого дефект присутствует в ИТ-системе. Со временем даже незначительные дефекты приводят к серьезным потерям

[1] *Poppendieck M., Poppendieck T.* Lean Software Development: An Agile Toolkit. 2003. ISBN 978-0321150783.

Как видно из таблицы выше, практически все виды потерь изначального списка, взятого из дисциплины бережливого производства, находят свое отражение в области информационных технологий. С момента публикации первых сведений о системе производства компании «Toyota» и по мере осмысления изложенных базовых идей многим последователям приходило в голову расширить оригинальный список. Так, разными авторами предлагалось добавить такие виды потерь, как:

- управленческие расходы (по сути, все, что делает менеджмент, а не рабочие);
- продукция или сервисы, не соответствующие ожиданиям и потребностям клиентов (что перекликается с классическим определением понятия качества);
- неиспользование творческого и интеллектуального потенциала сотрудников;
- неиспользование ресурса работников для улучшения процессов и технологий;
- недостаточное обучение персонала;
- использование некорректных метрик либо неиспользование измерений вовсе;
- неэффективное использование информационных систем (некачественная автоматизация, а также потери, связанные с непродуктивным использованием информационных технологий, как то: игры и общение в социальных сетях в рабочее время).

Разумеется, при наличии фантазии список видов потерь можно сделать достаточно большим – главное, не забывать про базовые принципы, лежащие в основе понятия «потери», а также помнить про влияние на практике каждого из видов потерь на принимаемые управленческие решения. Говоря про принципы, наиболее часто встречается следующий: потери – это то, за что клиент не заплатил бы, если бы у него был выбор. Очевидно, данное утверждение слишком общее и вряд ли подходит для решения задачи отнесения той или иной выполняемой работы либо к полезной, либо к потерям, особенно в пограничных случаях. Например, является ли потерями предварительная проработка архитектурного решения для данной ИТ-системы? Является ли потерями интеграционное тестирование, выполняемое при сборке и компиляции различных исходных кодов и модулей?

На мой взгляд, для практического применения базовый принцип можно сформулировать более точно: потери суть действия, не являющиеся необходимыми для получения конечного результата, которых можно избежать или минимизировать, если иначе организовать технологический процесс.

Применение идей бережливого производства на практике можно описать следующей последовательностью шагов:

1) используйте специализированные инструменты для *выявления* потерь;
2) примените другие специализированные инструменты для *устранения или уменьшения* потерь;

3) повторите шаг 1;
4) ...
5) PROFIT!!!

В бережливом производстве используется множество интересных концепций, практик и инструментов. DevOps заимствует многие из них, такие как цепочка создания ценности и ее картирование (англ. Value Stream и Value Stream Mapping), быстрое устранение препятствий (Andon), постоянный и равномерный поток, одна задача в единицу времени, выявление и устранение узких мест и ограничений, непрерывное совершенствование, вытягивающая система, визуализация работ и др. Некоторые из них будут рассмотрены в главах 3 «Принципы» и 4 «Основные практики».

Сложности применения

При всей привлекательности идей бережливого производства, у организаций, пытающихся использовать данные принципы в своей работе, возникают затруднения. Даже если не рассматривать применение Lean в сфере информационных технологий, а посмотреть на более широкий опыт организации производства продукции, можно заметить, что идеи бережливого производства могут не давать ожидаемой отдачи потому, что требуется довольно существенная перестройка работы организации – не только и не столько с точки зрения применяемых практик или инструментов, сколько с позиции принципов. Такого рода изменения требуют иной корпоративной культуры, нежели та, что имеется на большинстве предприятий. Сотрудники должны разделять общие ценности, зачастую не похожие на те, что встречаются в «традиционных» компаниях. Рассмотрим пример: идущий по цеху рабочий замечает лужу машинного масла на полу возле одного из станков. В компании, пропитанной духом бережливого производства, такой сотрудник просто не может пройти мимо – он должен предпринять действия к устранению беспорядка, так как он понимает и разделяет ту точку зрения, что данная лужа может привести (точнее – скорее всего, приведет) к выпуску некачественной продукции либо замедлению производства. В аналогичной ситуации в обычной компании сотрудник, скорее всего, просто пройдет мимо, ведь он уверен, что в организации есть специально обученные люди, чья ответственность – следить за порядком, а в его круг должностных обязанностей данная задача не входит. Понятно, что привить всему персоналу иную, неочевидную культуру – сложная, затратная и долгая управленческая задача. Публикации в прессе показывают, что далеко не у всех желающих получается в принципе ее решить; у многих же решение занимает годы, если не десятилетия.

Интересна известная история попытки распространения практик Тойоты в совершенно иной среде, концерне General Motors. Вкратце события были таковы.

Среди всех предприятий по производству автомобилей GM, завод в городе Фремонт, штат Калифорния, был худшим как с точки зрения качества производимой продукции,

так и с точки зрения управления. Ситуация дошла до того, что сотрудники открыто выпивали и играли в азартные игры прямо на рабочих местах во время своей смены, в то время как менеджеры ничего не могли с этим поделать. В 1982 году завод был закрыт.

Примерно в то же время компания Тойота пыталась выйти на рынок США, для чего ей было необходимо иметь локальное производство. Наилучшим решением представлялось партнерство с уже имеющимся игроком – Тойота получает возможность быстрого старта на местном рынке, а партнер – доступ к технологиям Тойоты, в том числе управленческим. В 1984 году тот самый завод «Fremont Assembly» открылся снова, уже под названием «New United Motor Manufacturing, Inc.» (NUMMI). Некоторые сотрудники, включая бывших профсоюзных лидеров, остались теми же, что и раньше. Они прошли обучение в Японии, а руководство было дополнено ценными зарубежными кадрами. В короткие сроки завод стал лучшим в GM как по качеству производимой продукции, так и по культуре производства. Японцы, без преувеличения, сделали маленькое чудо.

Разумеется, историю успеха следовало тиражировать. Следующим заводом был выбран Ван-Найс (Van Nuys), схожий по проблемам с предприятием во Фремонте. Однако любые попытки что-то поменять и улучшить были полностью провалены, несмотря на привлечение опытных руководителей с уже успешного завода NUMMI. «*Слишком многое было различным на этих заводах, NUMMI и Van Nuys*, – вспоминали потом менеджеры GM. – *Однако в то время мало кто видел и осознавал масштаб отличий, особенно невидимых. Культура GM описывалась словами "моя хата с краю", в то время как завод NUMMI работал как единая экосистема, включающая сотрудников, руководителей, поставщиков, смежников, разделяющих общие принципы*».

Следующие 15 лет компания GM потратила на анализ ситуации и принятие решения о целесообразности изменения культуры и производства. Еще 10 лет ушло на попытки такие изменения выполнить. В 2009 году компания GM обанкротилась и была выкуплена правительством США. В 2010 году завод NUMMI был закрыт, однако Тойота осталась на североамериканском рынке и в настоящее время занимает 15–17% его общего размера по числу проданных автомобилей.

Не меньшие затруднения испытывают те, кто пытается применить Lean ради Lean, а не для решения имеющихся задач. Такой подход встречается во многих областях, не будем уделять ему много внимания. Заметим лишь, что, как и прочие управленческие принципы и инструменты, бережливое производство – способ достижения целей, которые необходимо предварительно сформулировать, а уже потом достигать теми или иными средствами.

Сложность применения принципов бережливого производства именно в сфере информационных технологий заключается в том, что в обычном ИТ-департаменте непросто найти что-то, хоть отдаленно напоминающее конвейер. В то время как именно с конвейером зачастую ассоциируются применяемые в Lean практики, такие как андон и точно-во-время (JIT, just in time). Действительно, если рассматривать отдел разработки программного обеспечения как отдельную, условно независимую штатную структуру, то в ней можно представить некий конвейер, похожий на жизненный цикл ПО. Однако данный конвейер не завершается предоставлением ценности конечному потребителю (так как ограничен рамками одного из отделов ИТ), а потому является непол-

ным. В отделе эксплуатации конвейер найти еще сложнее. Возможно, поэтому отдельные авторы предлагают в качестве такового использовать процесс предоставления ИТ-услуг – глубина данной мысли не поддается измерению для любого, кто знаком с основами сервисного подхода в ИТ.

По своей сути, работа в отделе ИТ неосязаема: делается что-то, что невозможно потрогать, а иногда и просто увидеть, оценить. Ровно то же относится и к результатам – работающим ИТ-системам или предоставляемым ИТ-услугам, в зависимости от точки зрения читателя. Неосязаемость «заготовок», «труда» и «результата» в ИТ разительно отличается от производства продукции на заводе.

Завершим рассмотрение сложностей применения Lean красивой аналогией, подсказанной упомянутыми выше М. и Т. Поппендик: если взять за пример ресторан, то работа по созданию информационных систем скорее похожа на придумывание рецептов шеф-поваром, а производство продукции на заводе ближе к изготовлению блюд по ранее разработанным рецептам. Работа шеф-повара заключается в предположении наиболее изящных, востребованных и вкусных блюд, поиске способа их изготовления, попытке изготовить и последующей проверке, в том числе цикличностью, методом проб и ошибок, либо постоянном улучшении получаемого результата. Изготовление же блюд персоналом ресторана уже ближе к конвейеру, производящему «продукцию» по предварительно составленному рецепту, включающему в себя и перечень необходимых ингредиентов, и технологию производства.

Таким образом, прямое применение принципов и идей бережливого производства является не таким простым, как того хотелось бы, особенно если учитывать специфику современных информационных технологий.

Agile

Основные сведения

Возникновение, идеи и принципы Agile были достаточно полно рассмотрены в разделе «Развитие гибких методов разработки программного обеспечения». Agile служит мощным фундаментом для DevOps – настолько заметным, что и про него можно время от времени услышать категоричное утверждение отдельных энтузиастов о том, что в DevOps, помимо Agile, ничего больше и не придумано (вспомним начало раздела про бережливое производство с аналогичным наблюдением). Как и с Lean, с Agile история аналогична – данное утверждение далеко от истины.

Следует заметить, что Agile по своей сути является декларацией принципов и ценностей, в то время как описание применения этих принципов с учетом данных ценностей содержится в производных продуктах – различных методологиях разработки программного обеспечения. Таких методологий на данный момент насчитывается не менее десятка, наибольшую известность из них получил Scrum.

Не погружаясь в подробности разных источников знаний и не ставя себе задачи разобраться, откуда что пришло, можно выделить ключевые идеи и практики Agile, которые чаще всего упоминаются в DevOps:

- организация самостоятельных и самодостаточных команд небольшого размера (до 10–12 человек), предпочтительно находящихся в одном помещении и работающих над ограниченной областью задач;
- итеративный процесс создания и тестирования программного кода (спринты) с выдачей готового к использованию результата по итогам каждой итерации;
- ведение списка функциональных и нефункциональных требований (бэклог), служащего основой для планирования работ очередной итерации;
- разбиение больших задач на небольшие части (истории), их оценка в условных единицах объема работы для последующей приоритизации;
- активное вовлечение представителей заказчика в работу команды;
- регулярные краткосрочные совещания внутри команды с обсуждением плановых задач, прогресса и имеющихся сложностей;
- регулярные ретроспективы, помогающие самообучению команды и улучшению ее работы.

Некоторые пункты данного списка будут более подробно рассмотрены в главе 4 «Основные практики».

Сложности применения

Несмотря на ажиотаж, сопровождающий Agile в настоящее время, применение гибких подходов к разработке программного обеспечения во многих случаях вызывает затруднения.

Во-первых, как было показано в разделе «Развитие гибких методов разработки программного обеспечения», Agile охватывает только часть цепочки создания ценности, что дает скромный суммарный результат.

Во-вторых, Agile не учитывает специфику и сложности *эксплуатации* информационных технологий, где двигаться итерациями не всегда возможно – по крайней мере, если применять данную методику «в лоб».

В-третьих, если, согласно Scrum, конечный результат работы команды на очередной итерации – это новый программный код, прошедший регрессионное тестирование, то работа команды будет сводиться к постоянному бегу по кругу (спринтам), день за днем, неделя за неделей, притом что моральное удовлетворение от такой работы у персонала будет все меньше. Действительно, оценить изящность примененных алгоритмов могут только члены команды, а разработанный программный продукт эксплуатируется другим подразделением, работающим по иным правилам. Некоторые компании сообщают о выгорании персонала после нескольких десятков итераций.

Заметим, что история Agile далеко не завершена, эта область продолжает свое развитие. Примечательно также то, что ключевые персоны осознают всю сложность текущего момента – через десять лет после публикации манифеста

они снова собрались, чтобы обсудить достижения и проблемы. Одним из итогов встречи стал перечень из двадцати пунктов – проблемы движения, которые не очень хочется обсуждать широким кругом. Среди них: явный коммерческий интерес многих основоположников; представление Agile в мире так, как будто это не бизнес, а независимое движение любителей программирования; замалчивание сложностей применения и негативных примеров; игнорирование ограничений области применения и активная пропаганда универсальности подходов; размытая и неподтвержденная декларация бизнес-ценности; невозможность масштабирования за пределы нескольких команд; накопление и умножение технического долга.

Все это не означает, что на Agile следует поставить крест. Напротив, есть все предпосылки использовать практически полезные наработки, чтобы с данной платформы, данного фундамента идти дальше – в DevOps.

Филипп Крюхтен (Philippe Kruchten), который был на памятной встрече, посвященной десятилетию Agile, весьма интересно подвел итог первым годам существования идеи[1]:

«*Движение Agile в некотором роде напоминает тинейджера: очень самоуверенного, постоянно смотрящегося в зеркало, не терпящего критики, заинтересованного только в своих дружках, отрицающего всю мудрость прошлых лет (только потому, что она из прошлого), изобретающего новые причуды и новый жаргон, временами дерзкого и высокомерного. Но у меня нет сомнений в дальнейшем взрослении, становлении более открытым миру, более мыслящим и потому более эффективным*».

[1] https://www.infoq.com/articles/agile-teenage-crisis.

Глава 3

Принципы

Полезно отделять принципы от практик. Разумеется, в эти два слова можно вложить разный смысл, поэтому необходимо условиться, что понимается под ними в данной книге. Словом «принципы» обозначим ключевые идеи, на которых базируется весь DevOps, без принятия и применения которых от DevOps остается 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. Мне нравится слово «практики» тем, что оно довольно точно отражает суть – это именно то, что должно войти в *практику работы* данной организации.

> *«Что касается методов, их может быть миллион и более, а вот принципов всегда лишь несколько. Человек, который освоил принципы, сможет легко выбрать свои собственные методы. У того, кто пробует методы, игнорируя принципы, непременно возникнут проблемы»*[1].
>
> *Хэррингтон Эмерсон (Harrington Emerson),*
> *американский инженер и бизнес-теоретик,*
> *пионер дисциплины научного менеджмента, 1911*

Все основные принципы, описываемые международными экспертами DevOps, приведены в этой главе. Рассмотрению практик посвящена следующая глава.

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

[1] https://www.goodreads.com/quotes/346365-as-to-methods-there-may-be-a-million-and-then.

практических задач появляются новые издания, достаточно полно рассматривающие поток с прикладной точки зрения[1].

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.

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 создается 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 четким (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 между участниками 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 ее важными подробностями. Возможно, будет полезно добавить названия задействованных исполнителей. Нелишним также станет указание мест, где накапливаются очереди объектов, ожидающих обработки, а также мест, 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,

[1] Можно рекомендовать следующие:
- *Rother M., Shook J.* Learning to See: Value-Stream Mapping to Create Value and Eliminate Muda. Lean Enterprise Institute, 2009. ISBN 978-0966784305.
- *Martin K., Osterling M.* Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation. McGraw-Hill, 2014. ISBN 978-0071828918.

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. Еще хуже дело обстоит с показателем %C/A, так как его значение для каждого шага, как правило, совершенно неизвестно и может лишь быть 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 приведен на рис. 3.1.

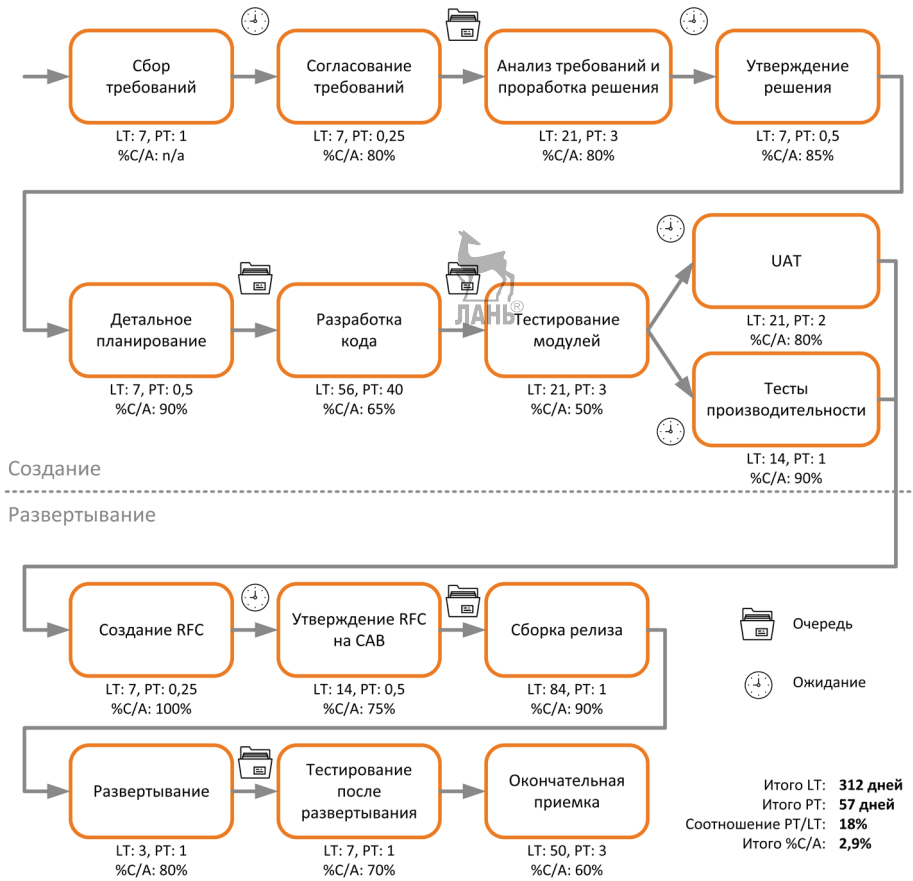

Рис. 3.1 ❖ Пример карты потока создания ценности

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 приведенном 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. Еще хуже дело обстоит с показателем %C/A, если в организации есть привычка отправлять обратно на предыдущие шаги задачу, которая была сделана неточно, 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 дает 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[1], 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

[1] https://www.tocinstitute.org/theory-of-constraints.html.

здесь появляются возможности применения инструментов и практик DevOps, изменяющих существующее положение вещей.

И наконец, в-четвертых, осознание потока создания ценности позволяет реализовать одну из основных идей DevOps – плавное и равномерное течение (англ. Flow)[1] работы от участка к участку, позволяющее создавать результаты постоянно, ритмично, без лишних задержек и с оптимальной загрузкой ресурсов.

Конвейер развертывания

Понимание потока создания ценности – необходимый и важный шаг на пути к DevOps. Однако работа с потоком «на бумаге» не дает столь значительных результатов, как ожидается. Предпосылки, изложенные в разделе «Истоки», позволяют сделать следующий важный шаг: построить конвейер развертывания (англ. Deployment Pipeline). 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 серьезном пересмотре. Помочь такому пересмотру призван конвейер развертывания, под которым понимается максимально автоматизированное сопровождение изменения по всем шагам потока создания ценности, начиная с момента «Разработка завершена» вплоть до состояния «Развернуто в среде эксплуатации».

Работа конвейера может быть проиллюстрирована схемой на рис. 3.2.

Конвейер автоматически запускается после того, как разработчик разместит в системе контроля версий новую часть программного кода, при этом фиксируется, кто, когда и какое изменение внес. По факту новой записи автоматически создается необходимая временная тестовая среда, в которой последовательно запускаются заранее разработанные тесты. Логика размещения тестов проста: проверки, обеспечивающие выявление большинства потенциальных ошибок, располагаются максимально ближе к началу конвейера. Все тесты, требующие ручного труда (если таковые есть), 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

[1] Английские слова «Stream» и «Flow» зачастую переводятся на русский язык одинаково как «поток», в то время как существует пусть сложно различимая, но все же разница между этими понятиями. Для ее подчеркивания в настоящем издании «Flow» переводится как «течение».

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.

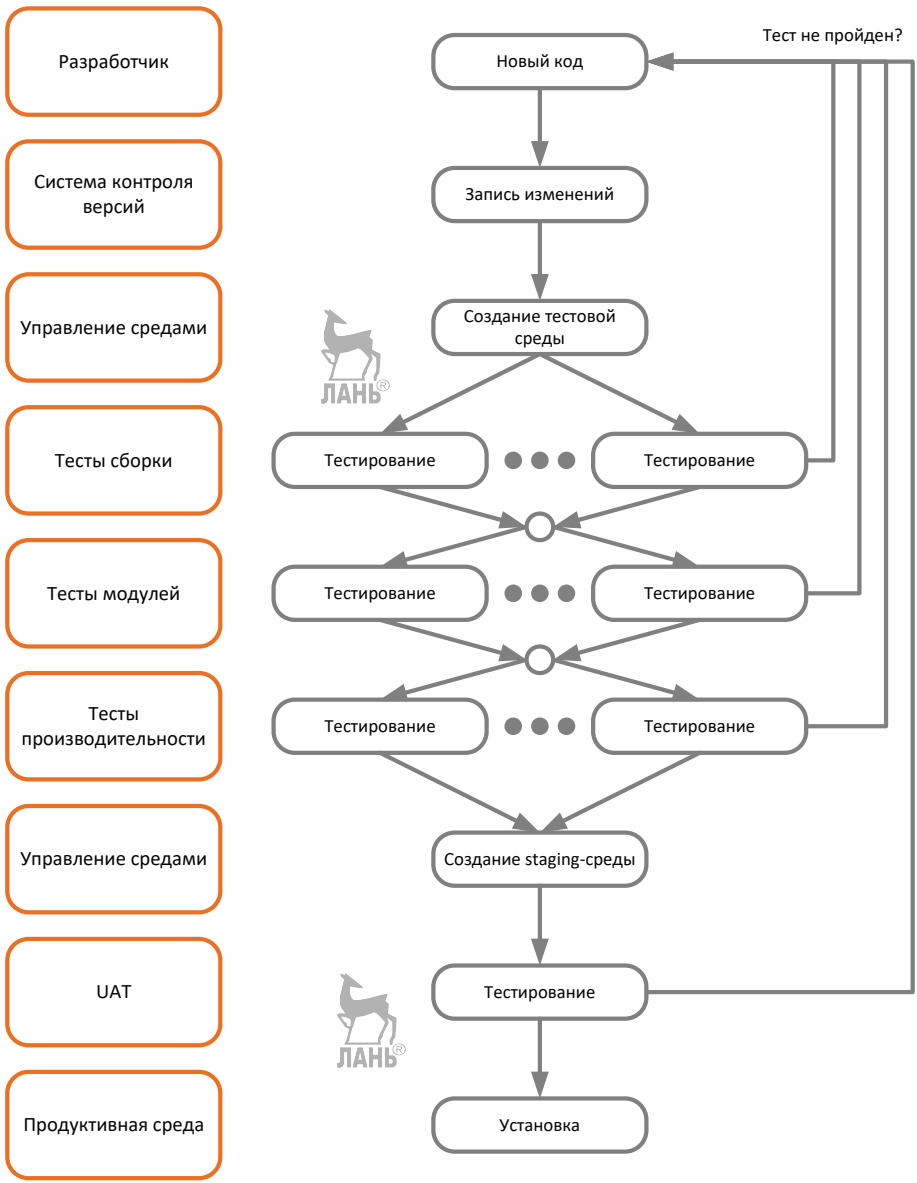

Рис. 3.2 ❖ Работа конвейера развертывания

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 счет максимально возможной автоматизации каждого из шагов. Наконец, в-четвертых, 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 развертывания 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 придется 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 оставшихся тестов.

Многие полагают, что название «конвейер» взято по аналогии со сборочным конвейером, например завода по производству автомобилей. Другие считают, что английское слово «pipeline» соотносится с жидкостью или иной субстанцией, текущей по трубам, и конвейер развертывания должен следовать такой аналогии. Оба приведенных мнения ошибочны.

Как поясняют авторы термина Д. Хамбл и Д. Фарли[1], оригинальная идея заключалась в аналогии с конвейеризацией, применяемой в современных процессорах, где повышения производительности сложно добиться только за счет увеличения тактовой частоты. Используемое архитектурное решение – параллельное выполнение инструкций, которые изначально поступают последовательно. Для этого процессор должен «догадываться» о результатах обработки в параллельном потоке, «надеясь», что они будут такими, как необходимо для выполнения вычислений в данном потоке. Если же нет – результаты вычислений будут отброшены. Потерянное из-за такой «догадки» время с лихвой компенсируется ускорением для тех случаев, когда «догадка» была верной.

Таким образом, правильно выстроенный конвейер развертывания позволяет сделать работы по разработке и тестированию независимыми друг от друга во времени: предполагается, что тестирование будет успешным, поэтому можно приступить к следующей единице работы. Такая же логика применяется и к выстраиванию параллельно работающих тестов.

С построением конвейера развертывания связаны еще три важных для DevOps понятия: непрерывная интеграция, непрерывная поставка и непрерывное развертывание (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 еще 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 еще и долгое время, то процесс сборки сам по себе является трудоемкой задачей, занимающей несколько недель. Действительно, необходимо учесть все изменения, сопоставить их друг с другом, обновить тесты с учетом изменений и сопоставления, переписать частично или полностью некоторую уже разработанную функциональность, и все это повторять до тех пор, пока новый код не будет приведен 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 пре-

[1] *Humble J., Farley D.* Continuous Delivery: Reliable Software Releases Through Build, Test, And Deployment Automation. 2011. ISBN 978-0-321-60191-9.

вращении ее 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 несет небольшой риск, но тут же запускают 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 одноименной книге, расширяет идею непрерывной интеграции: каждое сохранение изменений программного кода в системе контроля версий запускает не только процесс сборки, но и весь конвейер развертывания. Таким образом, все изменения, не прошедшие полного тестирования, не принимаются и требуют немедленного исправления. А все безошибочные изменения приводят систему к состоянию полной готовности развертывания в среду эксплуатации.

Непрерывное развертывание заключается в переходе от состояния «система всегда готова к развертыванию с учетом всех выполненных изменений» к состоянию «любое изменение незамедлительно разворачивается в среде эксплуатации». Переход к непрерывному развертыванию, 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 ее активация может быть выполнена дополнительно через программные настройки, или флаги, тогда, когда это будет нужно, скажем, отделу маркетинга. Подобная практика работы называется теневыми релизами (англ. Shadow Release) или темными запусками (англ. Dark Launches).

В любом случае, в основе данных практик – тот самый конвейер развертывания, описанный выше.

Все должно храниться в системе контроля версий

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

DevOps, как и во многих других областях, расширяет применение таких систем. Речь идет 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 развертывания, артефактов, используемых библиотек, создаваемой

документации, конфигурационных файлов, даже средств разработки, компиляторов, IDE и прочих инструментов. Перед каждым элементом приведенного списка уместно поставить дополнение «всех»: всех тестов, всех скриптов и т. д. Исключение делается только для двоичного кода, являющегося результатом компиляции программы, по следующим соображениям: обычно код занимает значительное место (что существенно, если он пересоздается 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 (рис. 3.3). Разумеется, применение такого принципа требует изменения культуры работы с информацией и конфигурациями.

Рис. 3.3 ❖ Система контроля версий

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 гарантированно рабочее состояние с минимальными трудозатратами. Еще одна появляющаяся возможность, пусть и менее значительная, – разрешение любому участнику команды свободно удалять ненужные более файлы и документы без риска случайной потери важной информации или наработок. Известно, что по мере развития проекта количество сопровождающих его файлов увеличивается, равно как и число вносимых в них изменений. Чистка мусора – занятие рисковое, если только нет постоянно создаваемой контролируемой копии.

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 создается 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 учетной 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 развертывания.

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.

> Распространено заблуждение, согласно которому полный DevOps наступает тогда, когда разработчики получают административные полномочия в среде эксплуатации, что размывает ответственность и нарушает стабильность системы.
>
> В действительности же можно утверждать, что администраторы урезаются в правах на среду эксплуатации, так как отныне им непозволительно что-то менять иначе, чем через полностью контролируемые скрипты.

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 т. д.

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 прак-

тика 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 еще дальше: автоматизированные системы регулярно изменяют административные пароли для доступа к разным средам, не сообщая новых паролей ИТ-сотрудникам. Таким образом обеспечивается отсутствие несанкционированных изменений в среде эксплуатации, хотя это правило действует для любых сред: разработки, тестирования, стабилизации и иных.

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 т. д. Однако в DevOps все совсем не так.

Один из ключевых принципов: работа завершена не тогда, когда кто-то сделал свой объем, 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 завершенной (рис. 3.4).

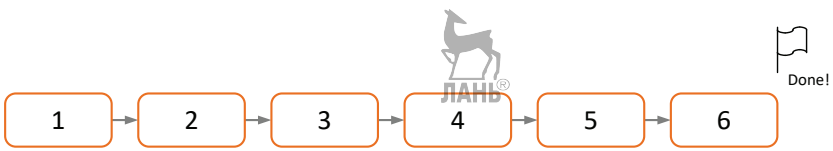

Рис. 3.4 ❖ Определение завершения

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 отдельные участки работы («*к пуговицам претензий нет?*») размывается, заменяясь коллективной ответственностью за общий результат команды («*костюм должен сидеть*»).

Радикально настроенные адепты DevOps настаивают на более жестком 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 развертыванию выполнены автоматически.

Краткое резюме главы

Для подведения итогов главы 3 «Принципы» вспомним данное в начале раздела описание этого понятия, как оно используется в настоящей книге. Итак, принципы – это ключевые идеи, на которых базируется DevOps, без принятия и применения которых от DevOps остается совсем мало смысла.

Действительно, без понимания, принятия и использования потока создания ценности, конвейера развертывания, полной системы контроля версий, автоматизированного управления конфигурациями и определения завершения можно сколь угодно долго упражняться с практиками DevOps, но отдача не будет существенной.

Глава 4

Основные практики

Обзор ключевых отличий от традиционных практик

В предыдущих разделах книги были рассмотрены предпосылки и истоки возникновения DevOps, его фундамент, связь с гибкой разработкой и бережливым производством, а также важные принципы, делающие DevOps возможным. К этому моменту у внимательного читателя книжки по обычной теме (программированию, анализу, архитектуре), вероятно, уже сформировалось бы понимание, что собой представляет рассматриваемая тема, зачем и кому нужна и что включает. С публикациями по управлению ИТ, будь то ITIL, COBIT или DevOps, такой фокус, к сожалению, не проходит. Читателю, скорее всего, понятны все изложенные ранее высокие материи, но очень хочется получить ответы на конкретные вопросы: что такое DevOps на практике? Что именно нужно делать и как?

К счастью, есть замечательный подход, очень наглядно иллюстрирующий суть явления через сравнение с условными «традиционными» практиками, – подчеркивание отличий поможет уловить самое важное. Такой подход я использовал на учебных курсах, семинарах и деловых играх по DevOps, проводимых в компании Cleverics, – полученный опыт позволяет сделать сравнение максимально лаконичным, отбросив все лишнее, но выбрав самые яркие примеры. Возможно, кому-то из читателей захочется большего – больше примеров, больше практик, больше дискуссий. Что ж, стремление к знаниям необходимо поощрять. Да что там, сам факт того, что вы дошли до этого места книги, уже достоин награды. А раз так, то, пользуясь служебным положением, я предлагаю вам, дорогой читатель, персональную скидку в размере 15% на любой учебный курс, семинар или деловую игру по теме DevOps от компании Cleverics. Нужно лишь сообщить об этом любому менеджеру по работе с клиентами[1], он сделает все остальное.

Мы же переходим к сравнению, которое, как было сказано выше, постараемся выполнить максимально кратко.

---

[1] Контактная информация: https://cleverics.ru/contacts.

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 еще лучше (стабильнее, безопаснее, быстрее и т. д.). Проверить такой большой релиз – отдельная задача, требующая внимательности, времени, привлечения множества специалистов. Все знают, что для любого релиза что-то обязательно пойдет 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 все 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 пойдет не так (особенно интересны случаи, когда релиз частично «установился», а частично – нет);
- ищут время в согласованном календаре изменений, позволяющем остановить работу системы – планово, если все пройдет хорошо, либо экстренно, если что-то пойдет 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 промежуточных результатов).

В зависимости от тщательности проработки каждого из пунктов данного списка длительность всего развертывания 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 проведенные изменения, сразу же настроить мониторинг новых и измененных компонентов. В случае каких-либо неполадок при развертывании конвейер автоматически прекратит распространение, откатит назад уже внесенные изменения и оповестит команду для принятия мер.

Компания Puppet совместно с некоторыми другими организациями каждый год на протяжении минимум последних четырех лет готовит и выпускает так

называемый отчет о состоянии DevOps («State of DevOps Report»)[1]. Отчет за 2017 год основан примерно на 5000 ответов респондентов из разных компаний разных секторов экономики; всего же за все годы было опрошено более 27 000 человек. Авторы отчета условно разделяют всех респондентов на три группы: ИТ-команды высокой, средней и низкой производительности. Наилучшим образом тезис «релиз – это рутина» иллюстрируется разницей в частоте развертываний: для ИТ-команд низкой производительности медианное значение частоты находится в диапазоне от еженедельно до ежемесячно, в то время как высокопроизводительные команды выполняют несколько развертываний ежедневно.

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 (неужели уже можно позволить себе называть ITSM классическим ИТ-менеджментом?) 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 еще не так хорошо вошло в русскую речь, потребуется еще 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, встает в очередь и дожидается очередного релиза: в счастливом случае ближайшего, но зачастую – подальше, через один-два квартала.

При использовании непрерывного развертывания в DevOps поставка новой функциональности в среду эксплуатации производится сразу же, как она разработана и протестирована. Пользователи ее 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 упоминалась в разделе «Конвейер развертывания», она 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-зеленые развертывания (англ. Blue-Green Deployments), для которых создаются две копии среды эксплуатации: «зеленая» и «синяя» соответственно. Переключение пользователей с одной

[1] https://puppet.com/resources/whitepaper/state-of-devops-report.

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[1] релизов (англ. Canary Releases), когда новая функциональность сначала становится доступной небольшому подмножеству пользователей. Убедившись, что все 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-четвертых, 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.

> Одного инженера компании Facebook однажды спросили: какова вероятность, что какой-либо конкретный пользователь Facebook является, сам того не зная, участником эксперимента? Инженер ответил:
>
> «*Определенно 100%. У нас одновременно и постоянно идет более двадцати экспериментов*».

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

Автоматизируется все, 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 остается 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;
- развертывание и тиражирование, включая настройку средств мониторинга.

---

[1] На протяжении нескольких веков рудокопы брали с собой в шахту клетку с канарейкой: эта птица очень чувствительна к метану и угарному газу и гибнет даже при незначительной их концентрации в воздухе, что является сигналом немедленно покинуть выработку и вернуться на поверхность.

Важное для DevOps повышение уровня контроля, описанное ранее в разделе «Все должно храниться в системе контроля версий», требует тотальной автоматизации всех ручных операций, в особенности– перечисленных выше.

Необходимые для работы конвейера развертывания среды создаются скриптами и автоматически, под управлением системы управления конвейером. Так же автоматически эти среды уничтожаются после использования, освобождая ресурсы. Конфигурирование элементов ИТ-инфраструктуры было подробно рассмотрено в разделе «Автоматизированное управление конфигурациями». 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, приемочных испытаний. Развертывание 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 развертывания. В таком случае событие, зафиксированное системой мониторинга, может привести к автоматическому откату назад уже развернутого 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 все еще не отключенные сервисы (подробнее данный вопрос будет рассмотрен в разделе «Эволюционирующая архитектура»).

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 еще – с длительным временем решения инцидентов. Для каждого инцидента необходимо произвести расследование, выполнить диагностику, найти и применить обходное решение – все это в подавляющем большинстве случаев выполняется вручную.

В DevOps все не так. В случае если инцидент связан с развертыванием, 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 в разделе «Управление ИТ-инфраструктурой как программным кодом».

Ошибки 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 ее 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.

Еще одна сложность, наблюдаемая на практике, заключается в невозможности объективно оценить размер очереди ошибок, которая имеет тенденцию к разрастанию. Десять ошибок – это еще допустимо? А пятьдесят? Пятьсот? Как сравнить ошибки разных приоритетов, значимости или ущерба? Может ли ошибка, находящаяся в очереди неделю, подождать еще? А месяц? Год? Принимая во внимание, что очередь ошибок находится в недрах какой-либо системы учета, увидеть и осознать ее – 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 стремясь управлять техническим долгом, большинство выявленных ошибок получает 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 соблюден.

Как и многие другие практики, такой порядок означает большую перестройку в планировании, приоритизации и выполнении работ, а кроме того – серьезные 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, принимаемое 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 ее результаты.

Процесс улучшается постоянно

Еще 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 описани-

ем. 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 регламентом 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 еще 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 развертывания, 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 чем состоит разница, можно ли ее сформулировать? Оказывается, можно – нижеследующая таблица перечисляет ключевые отличия.

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 приведенным характеристикам DevOps-культура сильно отличается от привычной, что, конечно, препятствует прямому и быстрому изменению стиля работы в обычных корпорациях. Приведенная 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, делающие невозможным либо крайне затрудненным использование традиционного функционального управления. В частности, автономный стиль, ориентация на продукт и сетевая организационная структура подталкивают к пересмотру способа группировки специалистов для достижения бо́льших результатов. На первый план выходят команды, а не структурные подразделения.

DevOps-команда – удивительная боевая единица. Она отвечает за небольшой, но четко 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 нее.

Команда не является способом объединить сотрудников на время, например на проект, – напротив, команда создается 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 остается 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 мини-коллектив, 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 иную загрузку. С другой стороны, такой подход увеличивает расходы на персонал (см. раздел «Уменьшение времени вывода на рынок»).

DevOps-команда является кросс-функциональной – 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 незавершенной работы.

Размер команды имеет важное значение. С одной стороны, ее не получится сделать слишком маленькой – небольшая команда не сможет стать кросс-функциональной, 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. Все это негативно сказывается на скорости работы.

Небольшой размер и необходимость кросс-функциональности выдвигают дополнительное требование к DevOps-командам: сотрудники должны быть максимально универсальными. Четкая 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 все устроено. Все это выравнивает загрузку и создает 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 большим, 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 серьезные 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 остается за кадром. Известны случаи, когда безобидные с точки зрения написанного текста комментарии вызывали бурю негодования, а сравнение с определенными 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, и 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 придется 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, притом 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.

Примером понятного для всех команд задания корпоративного стандарта может служить решение Джефа Безоса (Jeff Bezos) о переходе с монолитной на микросервисную архитектуру в 2001 году[1]. Отправленное им сообщение для технических специалистов гласило:

1) с настоящего момента все команды должны предоставлять свои данные и функциональность через сервисные интерфейсы;
2) команды должны обмениваться данными через такие интерфейсы;
3) другие способы коммуникации не допускаются;
4) можно использовать любые технологии;
5) все интерфейсы должны быть разработаны так, чтобы в любой момент стать доступными пользователям за пределами Amazon;
6) любой, кто не следует данным правилам, будет уволен.

[1] https://plus.google.com/+RipRowan/posts/eVeouesvaVX.

Описанные особенности DevOps-команд приводят к сложностям масштабирования – возникает необходимость взаимной координации работ на разных участках, что особенно важно при использовании общей ИТ-инфраструктуры. Наличие нескольких десятков команд подталкивает к формированию прослойки из менеджеров, которая в определенной степени идет вразрез с попытками увеличить скорость и снизить потери. Может показаться, что такие сложности непреодолимы, зато в привычной функциональной структуре они отсутствуют. Действительно, масштабировать традиционную организационную структуру вроде бы очень легко – можно добавить отдел, в нем назначить руководителя; можно поиграть в политику и поменять руководителей местами; можно при росте численности персонала соответствующим образом увеличить число уровней управления; можно ввести институт заместителей и совмещения должностей и т. д. Однако понятно, что подобные ухищрения обладают целым набором недостатков, скрывающих и маскирующих реальные проблемы взаимодействия, замедляющих работу и увеличивающих потери.

Итак, просуммируем ключевые особенности DevOps-команд (рис. 4.1).

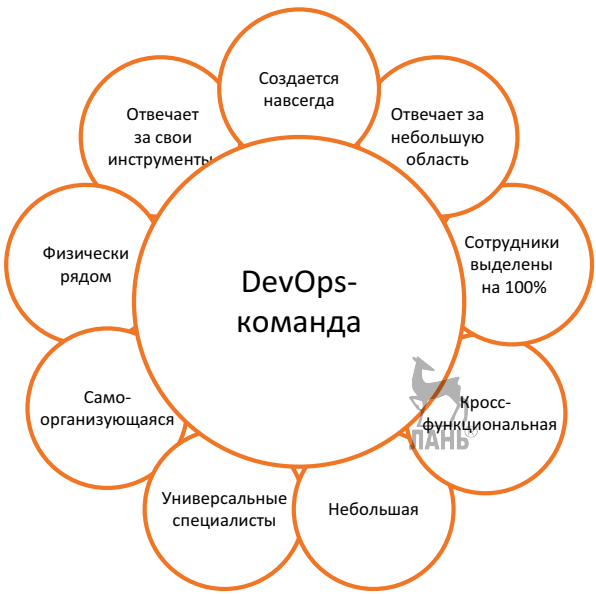

Рис. 4.1 ❖ Ключевые особенности DevOps-команд

Визуализация работы

Как уже упоминалось в разделе «Сложности применения» главы «Бережливое производство», работа в ИТ-отделе, в отличие от производства продукции, неосязаема – задачи нельзя потрогать руками, по «заготовке» не оценить степень

готовности, а по очереди задач на каком-либо участке не понять, завал там или рабочий режим. Такая неосязаемость мешает сотрудникам и руководителям в каждый момент времени знать ответы на ключевые вопросы, такие как:

- Сколько и каких задач взято в работу?
- На каком участке работа скапливается, образуя узкое место, не позволяющее эффективно работать всей остальной цепочке?
- Какие участки потенциально обладают недостаточной пропускной способностью и вскоре будут замедлять другие?
- Где ресурсы уже исчерпаны либо скоро будут исчерпаны?
- Какие задачи застряли так, что точно не успеют дойти до финиша в данной итерации?
- Что осталось сделать для задачи, которая не доделана до конца?
- Если мы не успеваем сделать всю взятую на себя в данную итерацию работу, то какую ее часть наиболее целесообразно постараться все же доделать, чтобы получить максимально возможный полезный результат?

По сути, речь идет об обеспечении течения потока (см. раздел «Поток создания ценности»), для которого вполне применимы принципы и методы теории ограничений. При этом рассматривается не только часть, касающаяся разработки, но вся цепочка, весь конвейер, до конца – разработанного программного кода, используемого потребителями. Обеспечить течение и помочь найти ответы на приведенные выше вопросы может какой-либо инструмент визуализации. Часто применяемые системы на основе списков, в которых задачи можно фильтровать и сортировать, к примеру по приоритетам, не до конца справляются с поставленной задачей, даже если над списками установить какую-либо надстройку в виде сводных отчетов или панелей управления (англ. Dashboard). Дело в том, что системы, основанные на списках, с трудом отображают именно течение задач от этапа к этапу. Более привлекательно выглядят так называемые доски канбан; простейший вариант такой доски приведен на рис. 4.2.

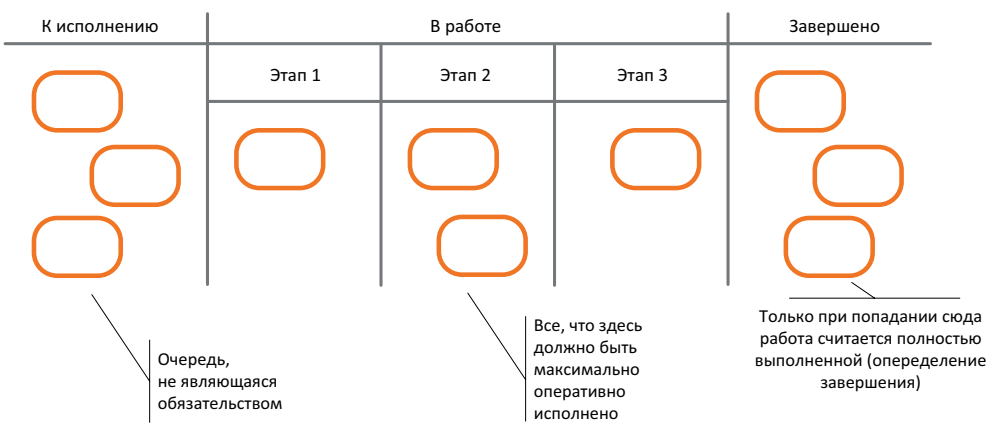

Рис. 4.2 ❖ Пример доски канбан

Очередь задач всей команды формируется на входе слева, в колонке «К исполнению» (англ. To Do). Далее слева направо расположены этапы работ, по которым постепенно двигаются задачи, – эта область часто именуется «В работе» (англ. In Progress). Справа находится финальная точка процесса, колонка «Завершено» (англ. Done). К такой относительно примитивной нотации можно добавить несколько важных замечаний из практики работы передовых DevOps-команд.

Во-первых, попадание задачи в очередь на входе, в противовес общепринятой практике, не означает принятия в работу. Задача в очереди означает буквально следующее: это хорошая идея, она *может* быть реализована, когда и если будет принято такое решение. Однако как только исполнитель первого этапа забирает первую задачу из очереди, он сразу же принимает и ответственность за ее скорейшее выполнение, при этом – от имени всей команды.

Во-вторых, приоритизацию работы необходимо выполнять только однажды, когда задача из очереди переводится на первый этап. В этот момент можно и нужно оценить и потенциальную выгоду, и требуемые ресурсы, и срочность, и многие другие параметры. Заметим, что существует множество методов определения приоритетов, и не все из них подразумевают детальный анализ приведенных параметров. Об одном из наиболее интересных способов управления очередью будет рассказано в разделе «Приоритизация задач». Так вот, как только решение о выполнении данной задачи принято, не выполнять ее на последующих участках уже нельзя – все, что начало свое движение по конвейеру, должно дойти до конца. Отмена одной задачи, чтобы добавить другую, недопустима, ведь незавершенная работа является одним из видов потерь, с которым необходимо всеми силами бороться. Следствие из данной практики – всем остальным участникам цепочки, кроме первого, не требуется тратить время на определение приоритетов поступающих задач, ведь приоритеты уже определены и не должны меняться.

В-третьих, внимательный читатель уже наверняка заметил, что выход предыдущего этапа является входом следующего. Таким образом формируется очередь каждого из этапов, что позволяет наглядно оценивать объем задач на разных участках.

И наконец, использование досок канбан позволяет выстраивать так называемую вытягивающую систему (англ. Pull System). Обычно тот, кто завершил предыдущий этап, «подкидывает» работу следующему по цепочке, стараясь как-то повлиять на принятие в работу, скорость выполнения и другие параметры своего смежника справа – по сути, «нагружает» того ответственностью. При вытягивающей системе, наоборот, исполнитель после окончания работы над предыдущей задачей сам принимает следующую и приступает к ее выполнению. Это позволяет обеспечить более плавное течение потока, уменьшить простой ресурсов и избавиться от необходимости тщательной координации

исполнителей – по сути, сильно снижая роль супервайзоров и других оперативных руководителей.

Следует отметить еще одно неожиданное применение досок канбан – они могут использоваться в качестве индикатора некорректного использования DevOps-подхода. Например, при неоптимально выстроенной приоритизации вход в цепочку быстро становится заваленным, приводя к беспорядку на всех остальных участках. В этом случае понятно, что заниматься оптимизацией потока не требуется, но необходимо разобраться с принципами выстраивания очереди и работы с ней, а также с фундаментальным подходом к управлению задачами: вытягивающая система или проталкивающая система (последнее, разумеется, противоречит самой идее использования канбана и бережливого производства). Второй пример – попытки применить доски канбан для подразделений, связанных с поддержкой и эксплуатацией, не меняя принципов их работы. В этом случае беспорядок также гарантирован ввиду большого числа отображаемых задач, делающих доску нечитаемой, а также довольно формального подхода к перемещению задач по доске – в системе изменили статус, задача переехала в соседнюю колонку, однако следующий исполнитель задачу фактически в работу не принял. Управления и течения при этом как не было, так и не появилось.

Просуммируем положительный эффект от всего вышеизложенного (рис. 4.3). Визуализация:

- позволяет выстраивать вытягивающую систему, что, в свою очередь:
  - улучшает течение потока работ;
  - уменьшает простой ресурсов;
  - уменьшает необходимость координации исполнителей;
- улучшает оценку общего объема задач в работе;
- улучшает понимание оставшегося объема работы и текущего состояния;
- улучшает приоритизацию задач;
- уменьшает количество передач в работе;
- помогает находить места неэффективности.

Ограничение числа задач в работе

Традиционная практика подразумевает поступление задач в работу из различных источников, притом асинхронно. Исполнитель, как правило, задействован в нескольких процессах, выделяя на каждый из них часть своего рабочего времени. Кроме процессов, задачи приходят от руководителя, от недовольных бизнес-подразделений, от коллег, которым нужно, как обычно, «всего лишь десять минут». Во многих компаниях применяемые средства автоматизации позволяют сделать чудесный список всех задач для конкретного исполнителя (рис. 4.4).

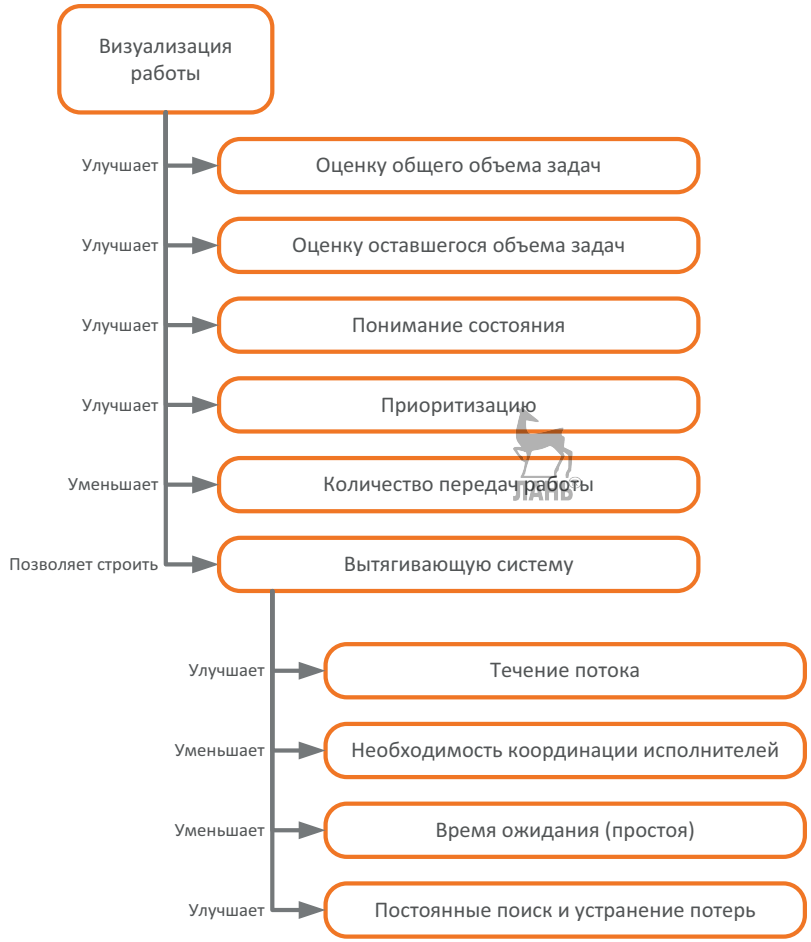

Рис. 4.3 ❖ Эффект от визуализации задач

Чудесного в таком списке много:

- он никогда не бывает полным, часть работ и поручений записывается и учитывается на бумажках, в блокноте, оперативной памяти исполнителя либо просто нигде;
- часть задач носит плановый характер, а другая – экстренный;
- вся работа из этого списка никогда не будет сделана в срок по миллиону различных причин, несмотря на согласованные приоритеты и установленные крайние сроки;
- многие задачи будут находиться в списке достаточно долгое время (месяцы, а то и годы), отчего актуальность списка снижается.

Мои задачи

Задача	Пояснение	Приоритет	Крайний срок
Обращение	Xxxxxxxxxxxxxxxxxxxxxxx	Нормальный	Вчера
Инцидент	Xxxxxxxxxxxxxxxxxxxxxxx	Высокий	Вчера
Запрос на изменение	Xxxxxxxxxxxxxxxxxxxxxxx	Плановый	Вчера
Наряд на работы	Xxxxxxxxxxxxxxxxxxxxxxx	Низкий	Вчера
Инцидент	Xxxxxxxxxxxxxxxxxxxxxxx	Критичный	Сегодня
Обращение	Xxxxxxxxxxxxxxxxxxxxxxx	Нормальный	Сегодня
Наряд на работы	Xxxxxxxxxxxxxxxxxxxxxxx	Высокий	Завтра
Наряд на работы	Xxxxxxxxxxxxxxxxxxxxxxx	Нормальный	Завтра

Рис. 4.4 ❖ Пример списка задач одного исполнителя

Наличие объемного списка задач приводит к хаосу. Этот хаос связан с частым определением приоритетов исполнителем («*за что мне сейчас браться?*»), частым переключением между задачами («*это не успеваю, возьмусь за вот это*»), частой сменой приоритетов из-за внешних факторов («*прилетела более срочная задача, эту пока оставлю*») и подобными неприятностями. В некоторых компаниях часто используемый способ приоритизации сводится к HiPPO (от англ. Highest Paid Person's Opinion) – по решению того, у кого самая старшая должность. Все формальные правила, равно как и здравый смысл, отходят на второй план.

Причина описанных бед кроется в многозадачности. Существует мнение, будто обычный человек способен делать несколько интеллектуальных дел одновременно, однако исследования последних десятилетий, равно как и практика, показывают, что это не так. ИТ-сотрудники в большинстве случаев в каждый момент времени делают только одну работу; если же пытаться выполнять несколько задач параллельно, то фактически начинается постоянное переключение между ними. Переключение само по себе требует времени, к которому необходимо прибавить как минимум время на приоритизацию и время на смену контекста. Замеры показывают кратное увеличение длительности выполнения задач в «многозадачном» режиме по сравнению с «однозадачным».

Рабочим способом борьбы с описанной практикой является ограничение числа задач в работе. С одной стороны, это звучит странно – получается, что исполнитель не будет принимать новую работу, которая входит в его должност-

ные обязанности? С другой – это действенный механизм обеспечения равномерного течения потока задач и выдачи результата в прогнозируемые сроки. Суть практики заключается в том, что на каждом участке потока создания ценности устанавливаются искусственные ограничения на число одновременно выполняемых задач или их общий объем (англ. WIP Limit, Work in Progress/Process Limit), в пределе – до одной задачи в единицу времени на каждом рабочем месте. Таким же образом ограничивается и общее число задач во всем потоке.

Подобная практика отлично поддерживает принцип вытягивающей системы, о котором уже много говорилось в данной книге. Действительно, при установленном ограничении у исполнителя предыдущего этапа работы просто нет возможности передать задачу на следующий этап: он лишь каким-либо способом информирует своего коллегу о том, что завершил свою часть работы. Следующий по цепочке доделывает свою текущую задачу, после чего приступает к следующей – ровно в момент, когда становится доступным для новой работы.

Следствием такой практики могут быть ситуации, в которых на отдельных участках потока нет никакой работы – они находятся в ожидании завершения предыдущих этапов. В обычном ИТ-отделе было бы принято решение заняться чем-нибудь еще, загрузить персонал какой-либо новой работой – нельзя же допустить, чтобы сотрудники простаивали! Они получают зарплату, а потому должны работать на сто, а лучше на сто двадцать процентов. В конце концов, максимальная утилизация ресурсов, в том числе персонала, – одна из важнейших задач любого руководителя. Но только не в DevOps.

Здесь не действует правило «Лучше делать хоть что-то, чем не делать ничего» – напротив, оно считается крайне вредным. Выполнение работы, у которой нет заказчика, – это потери. Обновление серверов до новых версий только потому, что вышли новые версии, – потери. Браться во время простоя за работу, которую затем бросить недоделанной, – потери. Вместо этого лучше воспринимать простой на одном участке как индикатор перегрузки на другом и сразу же принимать меры реагирования. Такие меры могут носить как оперативный характер (помощь коллеге), так и более системный (устранение возникшего узкого места потока). Основной акцент направлен на обеспечение течения потока, и в данном случае хорошо работает аналогия с течением реки или ручья: когда все хорошо и стабильно, скорость потока так же стабильна. Во время половодья на вход поступает слишком много воды, пропустить ее через русло не получится – река выйдет из берегов с неизбежными потерями собственно воды и ущербом для прибрежных территорий. Если же в обычной ситуации посередине реки вода мельчает, значит, где-то выше по течению появилось препятствие, запруда – ее необходимо найти и устранить, чтобы восстановить привычное, равномерное и предсказуемое течение.

Некоторые эксперты, такие как Д. Хамбл, рекомендуют устанавливать ограничения таким образом, чтобы они намеренно создавали неудобства. В момент, когда у кого-то в цепочке нет работы на входе и начинается простой

ресурсов, возникает страстное желание увеличить ограничения, установленные на предыдущих этапах потока, чтобы хоть какая-то работа была передана на исполнение. Ровно для борьбы с такими желаниями и фокусирования на устранении узких мест эксперты и рекомендуют задавать ограничения, привносящие «боль», но делающие явным ограничения системы.

Возвращаясь к вопросу утилизации ресурсов, можно отметить, что грамотно установленные и время от времени корректируемые ограничения на число задач в работе являются инструментом баланса между интенсивностью и продуктивностью работы. Дело в том, что между количеством задач в работе и средним временем выпуска (Lead Time) существует связь (рис. 4.5).

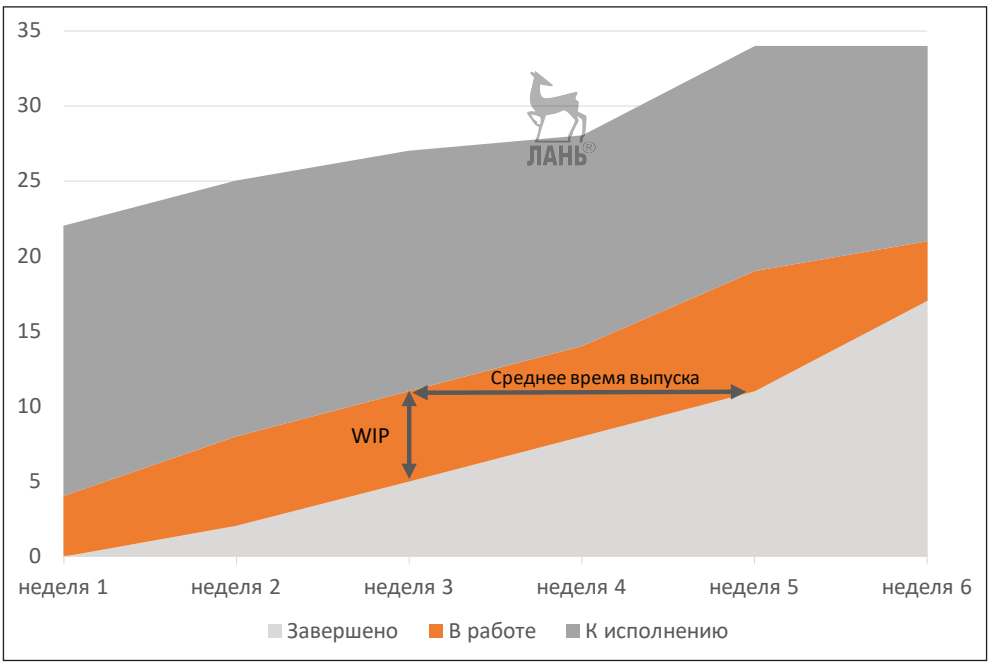

Рис. 4.5 ❖ Накопительная диаграмма потока

Важным аспектом является не только скорость выдачи результата, но и предсказуемость. Во многих случаях, несмотря на все усилия по оценке трудоемкости задач, бывает весьма сложно сказать заранее, сколько же в итоге данная работа займет календарного времени, то есть спрогнозировать время выпуска. На помощь приходят два инструмента: накопленная статистика о скорости работы DevOps-команды и управление ограничением числа задач в работе. Ужесточая ограничение, можно добиться меньшего времени выпуска в ущерб утилизации персонала, и наоборот. Таким образом, работа руководителей сильно изменяется.

Подведем общий итог последствий ограничения числа задач в работе (рис. 4.6).

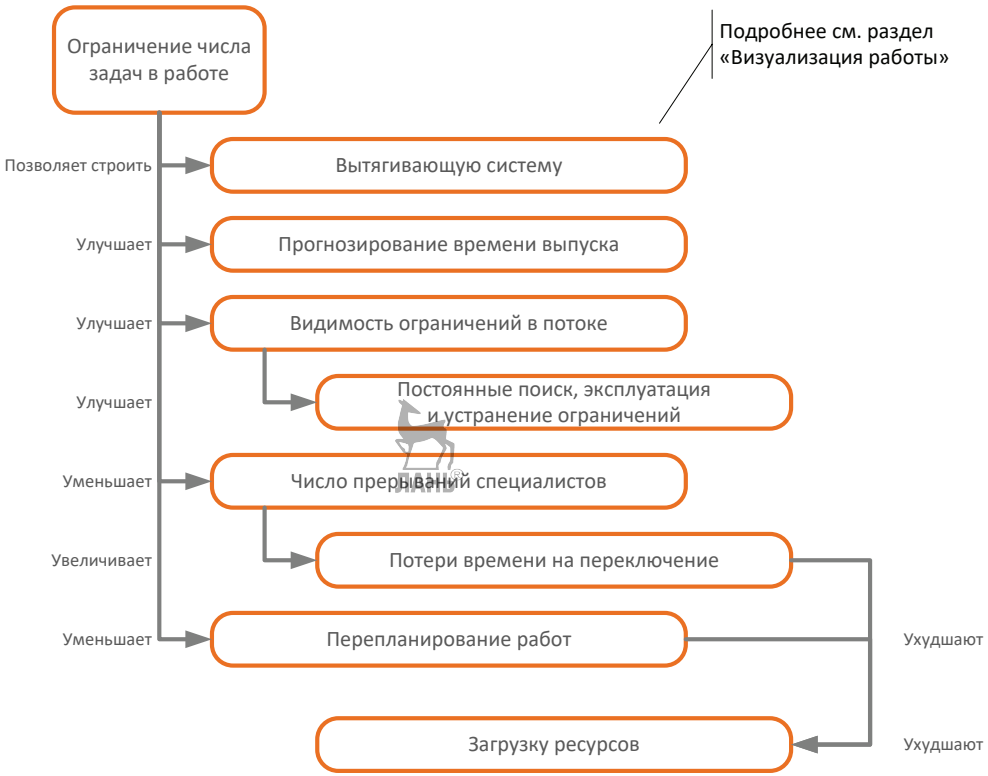

Рис. 4.6 ❖ Эффект от ограничения числа задач в работе

Уменьшение размера задач

Предположим, есть необходимость сделать, а затем проверить несколько одинаковых изделий. Первый вариант организации работ: делаем первое изделие, передаем на тестирование, в это время приступаем ко второму изделию, которое затем передаем на тестирование, и т. д. Второй вариант: делаем сразу все изделия, затем передаем их на тестирование. Часто встречается ошибочное суждение, что второй способ всегда эффективнее первого. Однако практика показывает, что эффективность зависит от общего размера партии, вариативности изделий, требуемой скорости получения готовых изделий, времени переналадки оборудования и других факторов, а значит – однозначного ответа нет.

В информационных технологиях вариант с меньшим размером пакета работ показывает лучшую эффективность, чем выдача результатов большими пор-

циями, по следующим причинам. Во-первых, большие партии редко бывают одинаковы по размеру, в отличие от максимально сокращенных, в пределе – до одного элемента работы в единицу времени (англ. Single-piece Flow). Таким образом, улучшается ритм выполнения работ, он становится более равномерным и предсказуемым на всех участках. Во-вторых, уменьшается время выдачи как первого результата, так и общее время выполнения работ за счет минимизации ожиданий в потоке создания ценности. В-третьих, небольшие партии сокращают общий объем задач в работе. В-четвертых, уменьшается число дефектов – в случае если допущена какая-либо ошибка, переделывать придется всю партию. Для небольшой партии потери от переделки будут ниже. Все это положительно влияет на ключевые характеристики DevOps: время выдачи, загрузку ресурсов и качество результатов (рис. 4.7).

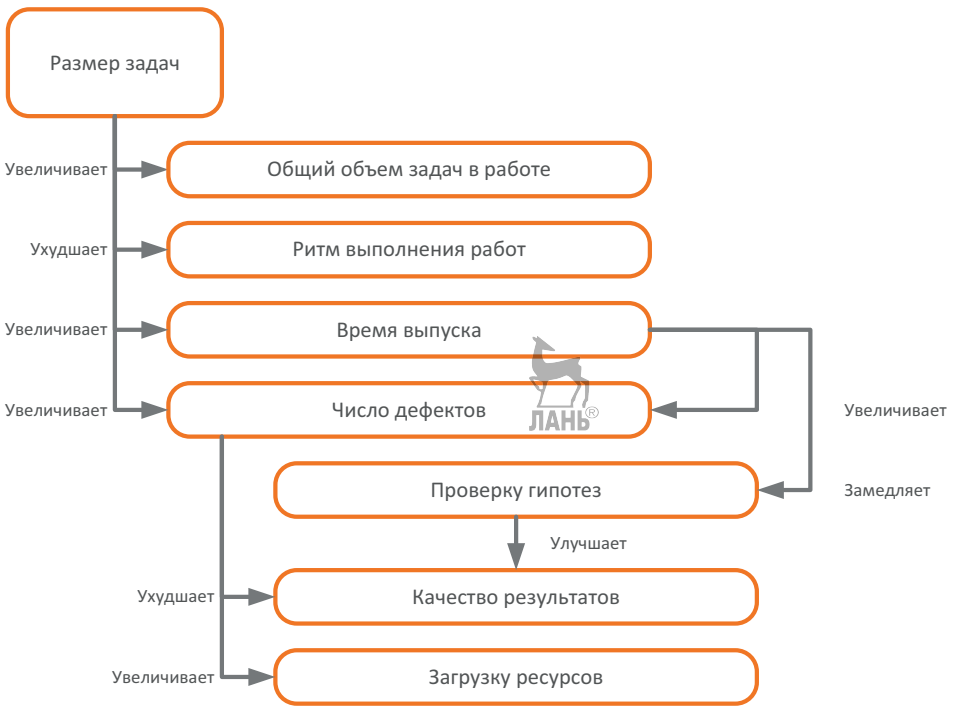

Рис. 4.7 ❖ Эффект от уменьшения размера задач

В обычной работе ИТ-отдела те самые «партии» бывает непросто обнаружить. Один из ярких примеров: выполнение программистами больших задач в течение нескольких дней, когда сохранение результатов в системе контроля версий производится лишь однажды и в самом конце. Рекомендуемый альтернативный способ – сохранять самостоятельные промежуточные результаты на всем протяжении выполнения работ, как минимум один раз в день. В хорошо

настроенном конвейере каждое сохранение запустит следующие шаги, например тестирование, что предоставит раннюю обратную связь и упредит дефекты. Ввиду небольшого объема каждого изменения выявленные дефекты будет проще устранить.

Выполнение операционных требований

В ИТ-подразделениях, где отделы разработки и эксплуатации являются отдельными друг от друга, вопросы обеспечения требуемой заказчику функциональности, как правило, находят свои ответы. Разработчики заинтересованы в том, чтобы заказчик описал то, что он хочет получить, и готовы реализовать данные требования. Отдел эксплуатации, в свою очередь, заинтересован в нормальных, рабочих отношениях с пользователями, что может быть достигнуто только в случае, если они могут выполнять свои задачи, используя созданную ранее функциональность.

Традиционной проблемной областью являются так называемые нефункциональные требования (англ. NFR, Non-Functional Requirements): доступность, надежность, расширяемость, масштабируемость, обслуживаемость, безопасность и подобные (рис. 4.8). Во многих из них в большей степени заинтересован отдел эксплуатации, ведь именно он сражается с инцидентами, устраняет корневые причины их возникновения, сталкивается с растущей пользовательской базой, взаимодействует с недовольными потребителями... Все это – при строгих ресурсных ограничениях, которые ослабляются медленнее, чем скорость увеличения требований к ИТ-эксплуатации. В то же время отдел разработки может фокусироваться в первую очередь на функциональных требованиях, в определенной степени игнорируя прочие.

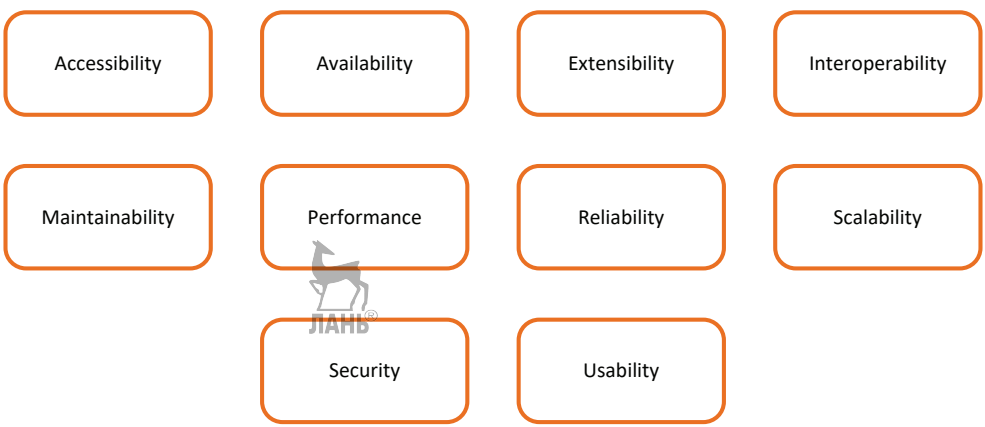

Рис. 4.8 ❖ Традиционный набор слов, труднопереводимых на русский язык, составляющих совокупность нефункциональных требований

Обычное решение этой проблемы для организаций, работающих по водопадной модели, – попытки вовлекать представителей ИТ-эксплуатации на ранних этапах разработки программного обеспечения. Слово «попытки» применено не случайно, так как лишь немногие компании добились существенных успехов в таком привлечении. Решение, которое применяют адепты гибкой разработки, – приоритизация нефункциональных требований наравне с функциональными, а именно: тем же порядком, с той же важностью, с таким же контролем исполнения. Такая практика, безусловно, лучше традиционной.

DevOps идет еще немного дальше. Во-первых, расширяется роль владельца продукта. В Scrum таковой описывается как лицо, наиболее заинтересованное в результатах либо представляющее интересы таких заинтересованных, однако с неким уклоном или ограничением, если хотите, в сторону реализации возможностей продукта (англ. Product Features). Эксперты DevOps предлагают рассматривать владельца продукта как заинтересованного в полностью работоспособной ИТ-системе, включая как функциональные, так и иные требования, что радикально меняет их значимость и смещает фокус команды в сторону работающего продукта, при этом работоспособность определяется не только выполнением заранее заданной функциональности.

Во-вторых, отдельные DevOps-визионеры настаивают на отказе от привычного наименования «нефункциональные требования», носящего негативный оттенок второстепенности или меньшей значимости, и замене его на «операционные требования» (англ. OR, Operational Requirements[1]).

В-третьих, предлагается полностью пересмотреть подход к обеспечению работоспособности ИТ-систем. В разработанных ранее системах делался большой упор на проектирование и построение с учетом требований к надежности: системы должны как можно реже выходить из строя. Для выполнения жестких требований применялись дорогостоящие специализированные программные и аппаратные решения, включающие резервирование, дублирование, горячие замены и подобное, как правило, от известных производителей, на закрытых и патентованных технологиях, с долгосрочными и недешевыми контрактами на поддержку и сопровождение. В DevOps же основной упор с обеспечения надежности смещается на обеспечение устойчивости: система должна иметь способность самостоятельно обнаруживать сбои восстанавливаться без существенной потери производительности и без влияния на потребителей. Сама система строится на основе большого количества относительно дешевых и легко заменяемых компонентов с добавлением технологий распределенного хранения данных, параллельных вычислений и подобных, предпочтительно на основе продуктов с открытым исходным кодом. Практика намеренного и безостановочного внесения хаоса и разрушений в среду эксплуатации упоминалась в разделе «Устранение хрупкости».

---

[1] Владеющие английским языком читатели могут оценить придуманную для иллюстрации важности операционных требований присказку: «…*OR this doesn't deployed into production*».

Таким образом, становится очевидно, что работа с операционными требованиями строится совершенно иначе. Более того, интересной и модной практикой стала демонстрация времени бесперебойной работы и состояния систем в виде, понятном конечным пользователям, как приведено в рис. 4.9.

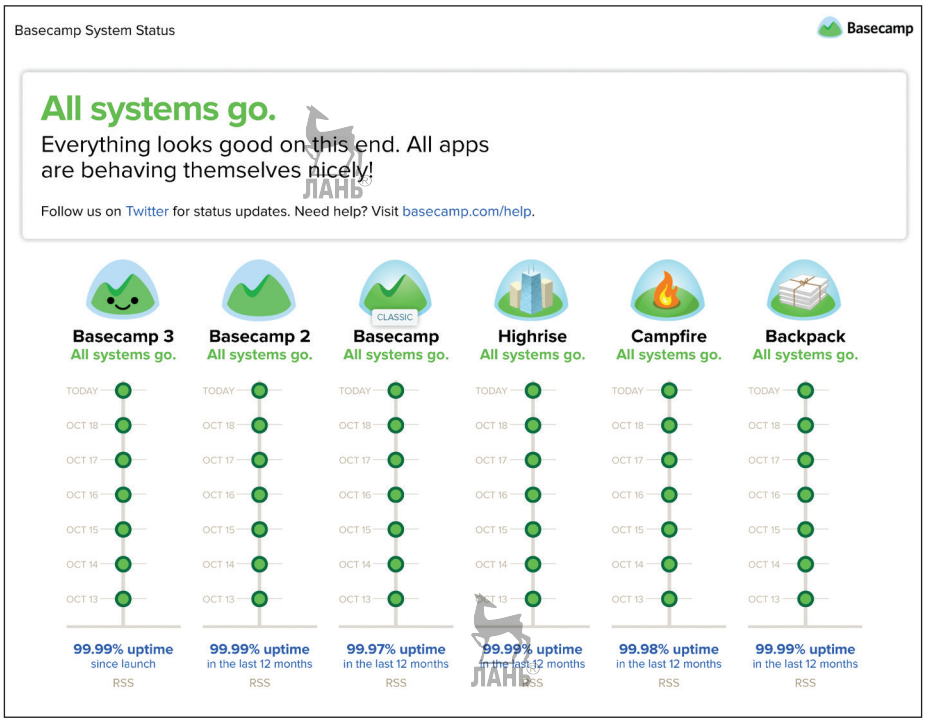

Рис. 4.9 ❖ Пример общедоступной страницы в интернете, отображающей актуальный статус систем компании Basecamp[1]

Такого рода страницы повышают доверие между DevOps-командой и пользователями и демонстрируют достижения не только на этапе разработки, но на протяжении всего цикла эксплуатации программного обеспечения. Дополнительно к этому единый источник информации о текущем состоянии позволяет упреждать массовые звонки, письма и сообщения пользователей в случае сбоя. И наконец, страницы демонстрации доступности являются прямой аналогией известных табло «146 дней без аварий», размещаемых на производствах, – такая наглядная информация повышает моральный дух работников, дарит им чувство безопасности и коллективной ответственности.

[1] https://status.basecamp.com/.

Раннее выявление и устранение дефектов

Наибольшие потери, связанные с информационными технологиями, возникают тогда, когда дефекты проявляются в среде эксплуатации – пользователи не могут выполнять свою работу, так как система недоступна, работает с перебоями, либо часть функциональности неисправна. Упреждению появления дефектов в среде эксплуатации в DevOps уделяется достаточно внимания, что уже описывалось в разделе «Конвейер развертывания». Однако при более пристальном анализе становится очевидно, что и внутри конвейера стоимость выявления и устранения дефектов увеличивается по мере продвижения от начала к завершению (рис. 4.10).

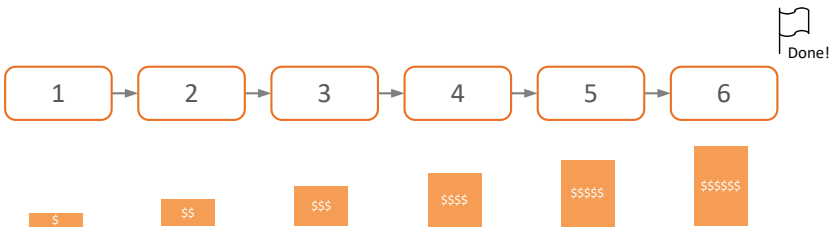

Рис. 4.10 ❖ Увеличение потерь при позднем выявлении дефектов

Действительно, для выявления дефекта на четвертом шаге мы уже потратили ресурсы в левой части, на шагах с первого по третий – к примеру, на создание требуемых тестовых сред и выполнение тестов. Эти ресурсы уже не вернуть, равно как и прошедшее время, что, безусловно, негативно отразится на времени выдачи результата. Особенно серьезной проблема становится в том случае, если не все части конвейера полностью автоматизированы – к примеру, если один из шагов требует ручного тестирования, то затраты на такое тестирование возрастают многократно. Получается, что чем раньше будут выявлены дефекты, тем лучше – как с точки зрения затрат, так и производительности конвейера. Возникает необходимость в ранней обратной связи, позволяющей вернуться на предыдущие шаги как можно ближе к началу конвейера. Одна из применяемых практик называется «сдвиг влево» (англ. Shift Left) – тестирование организуется таким образом, чтобы максимизировать выявление на ранних шагах наиболее частых ошибок. Это требует увеличения выполнения тестов на первых стадиях конвейера, что замедляет его начало, поэтому необходимо соблюдать определенный баланс.

Заметим, что наибольший выигрыш достигается в том случае, когда тестирование проводится максимально автоматизированно. Можно сказать, что в абсолютном DevOps-конвейере роль и работа тестировщиков меняется: их основной задачей становится не выполнение тестов, а их разработка. Это разделение естественным образом поддерживает следующий, пока еще справед-

ливый принцип: интеллектуальный труд остается за человеком, повторяющиеся рутинные операции можно передать машине.

Кроме того, скорейшему выявлению ошибок помогает практика, когда тестовые среды максимально точно соответствуют среде эксплуатации – насколько это возможно. Ситуации, при которых тестирование при выполнении конвейера было безупречным, однако в боевой среде приложение не работает должным образом, являются опасными и приводящими к потерям. Продолжая эту логическую цепочку дальше, следует стремиться, чтобы тестовые среды не только наиболее полно отражали среду эксплуатации, но и создавались теми же способами. Как было показано в разделе «Автоматизированное управление конфигурациями», в настоящее время это возможно.

Управляемые улучшения и инновации

Рассуждая о том, зачем же нужен DevOps, в разделе «Снижение технического долга», был рассмотрен важный вопрос постоянного накопления неоптимальности. Согласно естественному ходу дел, технический долг имеет тенденцию к увеличению, если не предпринимать специальных мер и действий. Аналогичное утверждение справедливо и для методов организации труда – процессов, процедур, договоренностей и т. д. Предоставленные сами себе, процессы начинают постепенно деградировать, сотрудники «срезают углы» не только в технических решениях, но и в порядке выполнения работ. Более того, динамика современной жизни подсказывает, что внешние факторы довольно часто меняются, и то, что отлично работало вчера – будь то программное решение либо процедура, – теперь работает уже не так эффективно.

Наконец, сами информационные технологии летят вперед семимильными шагами. Компания ThoughtWorks, весьма заметная на рынке построения эффективной разработки программного обеспечения, с 2010 года примерно каждые шесть месяцев публикует специальный отчет – так называемый «Радар технологий»[1]. В нем перечисляется более сотни позиций, сгруппированных в четыре области: техники, платформы, инструменты и языки. Например, среди техник в 2017 году присутствовали API-как-продукт, бессерверная архитектура, виртуальная реальность за пределами игр; среди платформ – Apache Mesos, AWS Lambda, PlatformIO; из инструментов – Airflow, HashCorp Vault, Terraform и др.; наконец, среди языков и фреймворков – Python 3, Elixir, Angular 2 и пр.

Каждая из наблюдаемых позиций авторами отчета относится к одному из четырех классов: смело применять, пробовать применять, оценивать, пока ждать. По многим пунктам дается пояснение. Таким образом, весьма трудозатратную работу по анализу новых технологий совершенно бесплатно для потребителей выполняет довольно квалифицированная команда предметных специалистов. Сравнивая между собой несколько недавних отчетов, можно

[1] https://www.thoughtworks.com/radar.

увидеть, как динамично развиваются (либо отмирают) новые технологии. Технологические возможности для поддержки основного бизнеса компании появляются постоянно, и многие из них потенциально могут принести серьезную отдачу.

Итак, технический долг необходимо сокращать, работу – улучшать, а новые технологии – осваивать. Современное ИТ-подразделение не может позволить себе заниматься этими важными задачами в фоновом режиме в свободное от «основной» работы время – с таким подходом получится в лучшем случае оставаться на месте, а в худшем (и более вероятном) – деградировать. Поэтому DevOps подразумевает постоянную управляемую практику улучшения и инноваций. Такая практика в разных компаниях может носить совершенно различный характер. Приведем лишь несколько примеров.

Некоторые компании начинают с выделения определенной доли рабочего времени на выполнение перечисленных выше задач, в литературе все чаще встречается словосочетание «налог в 20%» (англ. 20% tax). Понятно, что число взято с потолка и вряд ли может быть одинаковым для разных организаций, однако можно попробовать отыскать более взвешенные оценки. Например, модель SAFe предлагает командам двигаться так называемыми программными инкрементами (англ. Program Increment) длиной от 8 до 12 недель. Завершающая стадия каждого инкремента – итерация, посвященная инновациям и планированию. Справедливости ради стоит отметить, что в той же итерации отводится время и на завершение недоделанной работы в предыдущих спринтах, и на финальные интеграцию и тестирование, и на планирование следующего инкремента – таким образом, от условных двух недель остается лишь несколько дней, а выделенное время на инновации составит от 1,5% до 15% от полного времени текущего программного инкремента. Будем считать такую оценку нижней границей разумного. За оценку сверху можно взять слова Марти Кэгана (Marty Cagan), который в конце 1990-х годов пережил завал технического долга в компании eBay и затем изложил свои мысли в книге «Вдохновленные: как создавать продукты, которые клиенты полюбят»[1]. Он считает, что в отдельных запущенных случаях придется отдавать до 30% и более, но все команды, выделяющие менее 20%, вызывают у него недоверие. Достаточно характерно, что многие эксперты рекомендуют вводить запрет на выполнение какой-либо обычной работы во время, отведенное на улучшение, – ни программировать, ни тестировать, ни разворачивать приложения не допускается.

Другой практикой решения обозначенных выше задач являются блиц-улучшения (англ. Kaizen Blitz). В этом случае время на совершенствование может не планироваться заранее, а выделяться по необходимости. Также приветствуется привлечение сторонних по отношению к команде участников – это могут быть другие сотрудники компании либо приглашенные эксперты. Считается, что взгляд со стороны может помочь сдвинуть дело с мертвой точки и най-

[1] *Cagan M.* Inspired: How To Create Products Customers Love. 2008. ISBN 978-0981690407.

ти решения, которые изнутри команды не видны. Собственно блицы занимают от одного до нескольких дней и сфокусированы на устранении выявленных недостатков и узких мест. Таким образом, от каждого блица ожидается вполне определенный и ощутимый результат – в худшем случае это список действий к выполнению, а в лучшем – уже исправленные точки неэффективности.

Отдельные компании комбинируют два подхода – выделение времени и привлечение внешних специалистов, также возлагая большие надежды на передачу знаний и опыта внутри компании. Например, в компании Target отводится полный календарный месяц, на который в специально отведенное место (офис) перемещаются целые команды. К ним добавляются выделенные наставники, помогающие перестроить работу над уже имеющимися задачами так, чтобы добиваться большего теми же ресурсами за меньшее время. В компании выделены ресурсы на одновременное ускорение до восьми команд. Предполагается, что за проведенный таким образом месяц команда не только выполнит возложенную на нее работу, но и научится новым способам, приемам и техникам, а по возвращении будет способна передать свои новые знания другим сотрудникам.

Наконец, все большее распространение приобретает практика так называемых хакатонов (англ. Hackathon) – времени, специально выделяемого для освоения новых технологий и попыток создания новых продуктов и инструментов. Организаторы таких событий понимают и принимают, что не все созданное будет иметь завершенный вид, либо обладать коммерческим потенциалом. Однако все чаще встречаются примеры, когда на хакатонах разрабатывались прототипы новых пользовательских приложений, ставших впоследствии успешными продуктами, а также существенно пересматривались используемые внутренние технологии в попытке упростить все время усложняющуюся архитектуру, убрать жесткие связи, избавиться от накопленных зависимостей и т. д.

Обобщить практику постоянного улучшения и инноваций необходимо следующими двумя важными наблюдениями:

- ❍ такая практика должна быть управляемой, а не предоставленной самой себе;
- ❍ такая практика в вашей компании, скорее всего, будет мало похожа на аналогичные действия в других организациях.

Финансирование, помогающее инновациям

Наличие своевременного и достаточного финансирования – необходимое условие выполнения любой деятельности. Традиционно решения о выделении средств принимаются на основе средне- и долгосрочного планирования с применением бюджетных циклов. Такой подход прекрасно работал еще десять лет назад, однако в настоящее время создает препятствия для компаний, стремящихся стать лидерами инноваций. Что же с ним не так?

Начнем с того, что обычно финансирование выстраивается как циклический процесс – повторяющиеся, как правило, ежегодно итерации задают четкий ритм для планирования доходов и расходов. Само наличие ритма является положительным фактором, беда лишь в том, что этот ритм – другой, не тот, что нужен современной компании. Действительно, можно ли со всей определенностью предсказать отдачу через двенадцать месяцев от идей, которые еще даже не запущены в проработку? Можно ли уверенно рассчитать расходы на такой долгий период? Как было описано в разделе «Уменьшение времени вывода на рынок», уверенность в правильности выбранного направления зачастую может возникать лишь по ходу движения, а повороты на пути неизбежны. Совсем печально, если бюджетный цикл компании привязан к периоду налоговой отчетности. Возможно, отдельным подразделениям компании, отвечающим за бухгалтерский и налоговый учет, такая привязка очень удобна, но не следует забывать, что указанные подразделения не несут большой ценности для организации – они вторичны по отношению к целям, миссии, клиентам, продуктам и партнерам компании. Получается, что необходимость соответствия неким внешним требованиям определяет способ бизнес-планирования. Читатели, знакомые с азами управленческого учета, согласятся, что подобное искажение следует устранять – притом, что речь идет о практике, принятой на подавляющем большинстве современных предприятий.

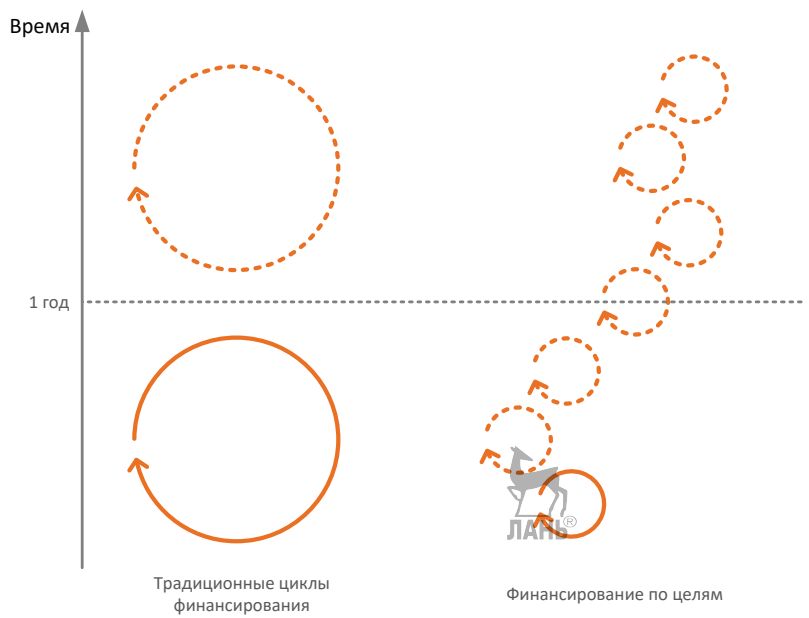

Рис. 4.11 ❖ Циклы финансирования не совпадают не только по времени, но и по направлению движения, а также редко могут быть синхронизированы между собой

Вторая принципиальная сложность возникает из-за повсеместного использования финансирования такой единицы учета, как проект: считается, что в подобном случае расходы можно тщательно спланировать и затем проконтролировать, а отдачу – предсказать. В то время как для большого числа современных инициатив проектное управление в целом имеет мало смысла, так как продукт, который организация стремится получить на выходе, не может быть заранее точно специфицирован, по мере выполнения работ много раз изменится, а после формального завершения такого «проекта» лишь начнет свой жизненный цикл, заключающийся в постоянной доработке и корректировке на основе нового полученного опыта использования и отклика клиентов. Эксперты DevOps не рекомендуют разрушать сложившиеся DevOps-команды, равно как и вовлекать в них сотрудников с частичной занятостью. Таким образом, более подходящей будет являться практика финансирования не проектов, а продуктов, что означает совершенно иной способ выделения средств и ресурсов.

Затронув тему ресурсов, нельзя обойти стороной еще одну типичную проблему – традиционные подходы к финансированию подразумевают жесткую конкуренцию разных подразделений или команд. Понятно, что ресурсы ограничены, однако DevOps в главу угла ставит принципы кооперации, совместной работы отделов и команд, свободного обмена знаниями и экспертизой. Если организация на уровне правил финансирования (то есть распределения ресурсов) задает необходимость сражаться с коллегами и смежниками, то не следует затем удивляться корпоративной культуре, в которой закрытость отдельных групп является нормой.

В прежние времена, когда сложность систем и скорость изменений были меньше, традиционные способы финансирования работали относительно хорошо (за исключением случаев, когда бюджетирование применялось как инструмент ограничения, а не приоритизации). Сейчас же создаваемые барьеры слишком мешают инновациям. Альтернативный подход, позволяющий получать высокую отдачу от инвестиций, – это создание стабильных продуктовых или сервисных команд и финансирование их на постоянной основе с определенной степенью свободы в выборе стратегии, способов реализации и приоритетов в своей зоне ответственности.

Разумеется, сказанное выше не означает отсутствия ограничений, к примеру, на расходы. Напротив – опыт зарубежных стартапов показывает, что в условиях жесткого учета и контроля расходов могут проявляться чудеса изобретательности и возникать новые, уникальные технические решения, на которые не способны выйти другие команды с иными принципами ресурсного планирования. Жизнь снова подтверждает многократно высказанную теорию о том, что неограниченного финансирования и бесконечного календаря все же недостаточно для создания конкурентоспособных продуктов, за которыми потребители будут выстраиваться в очередь.

Описываемый альтернативный подход, который DevOps в определенной степени заимствует из современных практик бережливого производства

уровня предприятия (англ. Enterprise Lean Management), подразумевает высокоуровневое определение долгосрочных целей, более точное планирование ближайших действий и постоянную подстройку краткосрочных планов для обеспечения верного направления движения.

В наиболее простом варианте, уже упомянутом выше, изменение принципов финансирования будет заключаться в переходе от проектного финансирования к выделению средств на каждую команду. Более сложным, но и намного более продуктивным может быть процесс, описанный Д. Хамблом[1]:

- шаг «Ориентация»: поиск и оценка идеи без затрат времени и ресурсов на проработку точной и сложной бизнес-модели;
- шаг «Исследование»: выделение фиксированного времени и некоего бюджета, сбор команды, разработка минимального рабочего продукта (англ. MVP, Minimum Viable Product);
- в случае если на предыдущем шаге продукт показал свою привлекательность – шаг «Эксплуатация»: сохранение команды, расширение финансирования, развитие продукта, поиск дополнительных возможностей.

Основной целью такого процесса является отбор интересных идей, инвестирование ограниченных ресурсов в некоторые из них с ожиданием, что большинство идей «не взлетит», но отдельные наверняка покажут значимый результат. Заметим, что по схожим принципам работают многие технологические бизнес-инкубаторы, где инновации поставлены на поток.

Понятие MVP, использованное выше, зачастую применяется совершенно некорректно. Многие считают, что MVP – это что-то сделанное «второпях и на коленке», некая минимальная функциональность, хоть как-то работающая без явных и заметных дефектов, чтобы продемонстрировать прототип инвесторам. Этакая ранняя альфа-версия продукта.

Однако автор термина MVP, Эрик Райс (Eric Ries), описывал его[2] как стратегию инвестирования минимально возможного количества ресурсов для получения через тестирование и обучение максимально возможного объема новой информации для принятия решения, основанного на объективных данных: продолжать в том же направлении, изменить курс или отбросить идею.

Марти Кэган (Marty Cagan) добавляет значимые характеристики MVP:

1) клиенты должны захотеть купить или использовать продукт;
2) клиенты должны суметь разобраться, как использовать продукт;
3) мы должны быть способны произвести полный продукт, когда и если будет принято такое решение.

Многие под MVP подразумевают лишь последний пункт, маскируя его функциональностью. В действительности все три важны: MVP – не часть третьего пункта (неполная функциональность), но часть каждой из трех приведенных характеристик.

[1] *Humble J., Molesky J., O'Reilly B.* Lean Enterprise: How High Performance Organizations Innovate at Scale. 2015. ISBN 978-1449368425.

[2] *Ries E.* The Lean Startup. 2011. ISBN 978-0307887894.

Приоритизация задач

Область, часто вызывающая затруднения, – приоритизация задач в очереди на входе в поток создания ценности. Традиционный подход призывает, прежде чем браться за работу, проанализировать задачи, оценить их, сравнить между собой и приоритизировать, получить одобрение или разрешение и лишь затем приступить к выполнению. Все перечисленные действия требуют времени и ресурсов – как правило, весьма значительных. При этом недостатков возникает сразу несколько. Во-первых, появляются серьезные временные задержки. Во-вторых, информация о задачах постоянно устаревает, поэтому чем дольше принимается решение, тем на менее достоверных данных оно будет основано. В-третьих, значимость перечисленных шагов в значительной степени преувеличена. И наконец, самое неприятное: долгий этап предварительной оценки служит катализатором создания гибридной модели, получившей название водоскрам-пад (англ. Water-scrum-fall) – команда считает, что работает модно и гибко, в то время как на самом деле движется весьма традиционно (рис. 4.12).

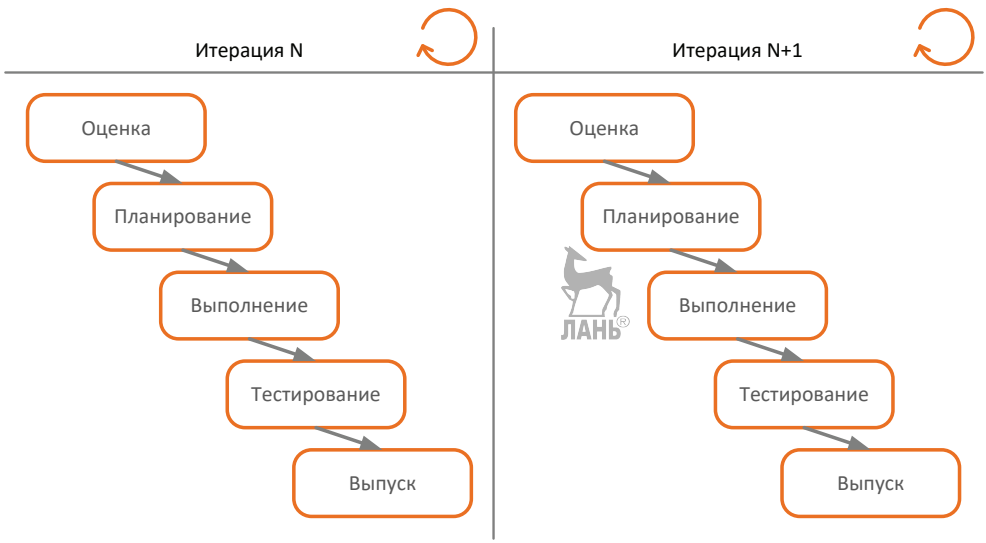

Рис. 4.12 ❖ Водопад, умело маскирующийся под итерации

Затруднения на первых шагах связаны с необходимостью выбора из общего пула той задачи, за которую следует взяться в первую очередь, для чего многие применяют набор критериев. Во-первых, требуется тем или иным способом оценить потенциальную выгоду – как минимум путем сравнения данной задачи с другими. Оценка выгоды дается нелегко: методологии вроде Scrum не сильно помогают в данном вопросе, скорее, ожидая поступления информации от заказчика или владельца продукта. Далее возникает желание оценить требу-

емые для выполнения ресурсы, но и здесь сложно предположить что-то определенное, особенно для задач, за которые команда берется впервые. Методики типа планирования через покер (англ. Planning Poker) упомянем, но детально и всерьез рассматривать не будем. Наконец, при определении приоритета очень опасно убирать из рассмотрения срочность – оценки выгод и ресурсов делаются не абстрактно в календаре, а на определенный момент времени, и результат требуется не просто когда-нибудь, а с возможной зависимостью отдачи от времени реализации, которую недостаточно описать мантрой «как можно скорее».

В 2009 году Дон Райнертсен (Don Reinertsen) предложил принципиально иной подход к приоритизации, назвав его стоимостью задержки (англ. Cost of Delay)[1]. Метод является относительно несложным, основан на экономической оценке значимости принимаемого решения и работает тем лучше, чем больше очередь работ. Первый шаг метода – определение ключевой метрики для данного потока создания ценности. Во многих случаях ей станет прогнозируемая финансовая отдача, однако возможны ситуации, где более значимым станет другой показатель. Определившись с единицей измерения, далее следует для каждой задачи рассчитать или оценить, что будет с данной метрикой, если задача останется в очереди, то есть будет задержана. Опыт Д. Райнертсена показывает, что большинство участников команд не представляет себе значение данного показателя, а попытки его оценить, как правило, дают существенную погрешность, поэтому автор метода настаивает на максимально точном, насколько это возможно, расчете, в том числе с учетом динамики во времени. Такой подход поднимает неудобные вопросы, на которые команда со временем научится отвечать.

Получив значение параметра, можно довольно легко сравнивать разные задачи между собой. В простейшем случае для задач с одинаковой длительностью выполнения в очередь должна поступать та, у которой стоимость задержки выше. В общем, более сложном случае удобно использовать производную метрику – стоимость задержки, разделенную на длительность (англ. CD3, Cost of Delay Divided by Duration). Ценность учета длительности выполнения заключается в возможности наглядной демонстрации, что при ограниченных ресурсах долгие задачи блокируют очередь для более мелких задач, задерживая потенциальную выгоду от них. Таким образом, сам принцип приоритизации поощряет как можно более мелкое разделение задач с сохранением их ценности, что ускоряет время выдачи полезного результата и обеспечивает более равномерную загрузку ресурсов.

Одно из преимуществ метода стоимости задержки состоит в простоте его использования для того, кто находится в самом начале потока создания ценности – именно там обычно возникают значительные потери времени, и именно

[1] *Reinertsen D.* The Principles of Product Development Flow: Second Generation Lean Product Development. 2009. ISBN 978-1935401001.

там ускорение даст наибольший эффект. Действительно, раз для каждой задачи определен единственный значимый показатель, то исполнитель, как только освободился, просто берет в работу следующую задачу с наибольшей стоимостью задержки (либо CD3). Тот же принцип можно использовать и далее по цепочке, однако для коротких итераций в одну-две недели на последующих этапах можно вовсе обойтись без приоритизации, доверяя выполненным ранее расчетам.

Второе значимое преимущество – экономически обоснованное принятие решений, прозрачное для всех участников. Не требуется сводить вместе несколько параметров, либо дожидаться утверждения вышестоящим лицом, либо применять метод HiPPO (см. раздел «Ограничение числа задач в работе»).

Наконец, третья, не заметная на первый взгляд особенность метода – помощь в активном освоении практики ограничения одновременного числа задач в работе. Действительно, при параллельном выполнении нескольких задач метрика будет давать наихудшую стоимость задержки, а потому при четком следовании методу никто и никогда не будет браться за несколько дел разом.

Однако следует помнить, что стоимость задержки – не еще один дополнительный параметр к уже имеющемуся набору. Напротив, он призван исключить все другие параметры, упростив принятие решений и сократив потери на самом старте потока создания ценности.

Постоянный поиск, эксплуатация и устранение узких мест

Рассмотренному в главе 3 «Принципы» потоку создания ценности всегда будут присущи ограничения. Работе с ними должно уделяться постоянное внимание. Состояние равномерного течения без задержек достигается не сразу и требует усилий. А это означает необходимость следующей регулярной практики: с помощью средств визуализации, а также искусственно созданных ограничений на размер задач в работе можно выявлять узкие места данного потока создания ценности. Среди всех известных узких мест требуется найти одно, которое создает наибольшие замедления, и далее работать только с ним.

Собственно, работа с узким местом состоит из двух направлений: во-первых, следует понять, как краткосрочно поменять правила работы так, чтобы наиболее полно и ценно использовать выявленное узкое место (эксплуатировать его). Например, ограничить задачи в потоке так, чтобы они, по возможности, не создавали нагрузки на выявленную точку ограничений, а также передавать в такую точку только приоритетные, важные работы. Во-вторых, необходимо найти способ устранить узкое место, избавиться от него. При этом стоит помнить об опасности вновь получить через некоторое время ограничение в том же месте, что связано с инерционностью системы и стремлением любых процессов вернуться к предыдущему, привычному состоянию.

После устранения выявленной точки ограничений можно отменить введенные ранее особые правила работы и перейти к поиску следующего наиболее значимого узкого места.

Краткое резюме главы

Объем любой книги ограничен естественными причинами. Объем данной книги, кроме того, ограничен сознательно – мало у кого в сегодняшних реалиях есть возможность тратить существенное время на чтение длинных текстов. Поэтому рассмотрение других DevOps-практик, коих в литературе и опыте работы разных организаций встречается великое множество, остается за кадром.

Примечательно, что, как и было объяснено в главе 2 «Фундамент», очень большое число практик унаследовано или позаимствовано из таких областей, как управление ограничениями, бережливое производство, непрерывное развертывание и других, давно существующих сфер менеджмента. Что, разумеется, не делает их менее пригодными инструментами решения управленческих задач в DevOps. В деле дальнейшего изучения таких практик читателю может помочь множество интересных, уже написанных книг; а завершающая глава данного издания – вопросы применения DevOps.

Глава 5

Вопросы применения

Область применения и ограничения DevOps

Содержание предыдущих глав книги, вероятнее всего, оставило у читателя ощущение некой сказки: вот было бы здорово в ней оказаться! И сотрудники в самоорганизующихся командах будут сплошь замотивированными, и бизнес будет вместе с ИТ-отделом искать дорогу к любимым клиентам и их деньгам, и ИТ-системы станут стабильными и антихрупкими, а изменения и релизы будут поставлены на поток вместе с уменьшающимся технологическим долгом… Выбранный ранее метод объяснения через сравнение условной традиционной практики с опять же условной DevOps-практикой по присущим ему причинам может выглядеть как подчеркивание преимуществ и сокрытие недостатков.

Вместе с тем я старался рассматривать все аспекты DevOps максимально беспристрастно. Как и было заявлено в самом начале книги, 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 определенных задач. Более того, как и другим подходам, ему присущи некоторые ограничения. Рассмотрим их, продолжая придерживаться трезвого и прагматичного взгляда.

Раз уж вы добрались до последней главы этой книги, то, скорее всего, дорогой читатель, жизнь вашего ИТ-подразделения в настоящее время устроена совершенно иначе. Масштаб требуемых перемен велик, а большие изменения и перестройки имеет смысл затевать только тогда, когда есть четкое понимание двух аспектов: выгоды и реализуемости. Рассмотрим их по порядку.

Начнем с того, что далеко не каждой организации в принципе следует задумываться про DevOps. Для начала отсечем 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. 5.1).

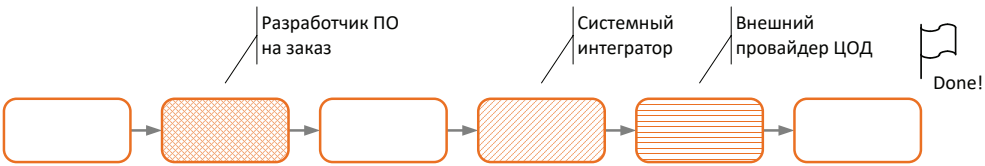

Рис. 5.1 ❖ Особые случаи, не рассматриваемые далее

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 партнерство, 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 (см. раздел «Уменьшение времени вывода на рынок»);
- 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.

Примером угроз, упомянутых в списке выше, может служить следующий факт: один миллион новых клиентов подключается к системе платежей Apple Pay каждую неделю[1]. Часть транзакционного дохода от их платежей за совершенно различные услуги, не относящихся к продуктам и сервисам компании Apple, теперь получает именно эта корпорация, но не банки. Масштаб потерь можно оценить через сравнение общего чис-

[1] http://fortune.com/2017/05/02/apple-pay-volume-up/.

ла активных карт Visa (порядка 2,5 млрд штук)[1] и активных аппаратов iPhone (0,7 млрд штук)[2]. Аналогичная история в то же самое время происходит с системой Android Pay. При этом ни Apple, ни Google не обладают банковской лицензией, не обязаны соответствовать жестким законодательным требованиям, не несут расходов на содержание отделений, филиалов и сети банкоматов. Зато они обладают мощнейшим финансовым ресурсом, передовыми технологиями и доступом к лояльной аудитории, что недостижимо ни для одной финансовой организации в мире.

Двигаясь в привычном им быстром темпе технологических компаний, они представляют угрозу для любого неповоротливого традиционного банка, привыкшего десятилетиями собирать обычным, как все другие банки, способом свой доход с расчетных операций, депозитов и кредитов.

Если для данной организации пункты приведенного 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 принесет большой пользы и совершенно определенно 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.

Перейдем к рассмотрению второго аспекта, реализуемости. Во всех ли компаниях можно «построить» DevOps? Мнение большого числа зарубежных экспертов сводится к положительному ответу. В его подтверждение часто приводится случай в подразделении по разработке встроенного программного обеспечения компании HP[3]. В этом департаменте трудится более четырехсот разработчиков ПО для принтеров, сканеров и многофункциональных устройств, сотрудники распределены по трем странам. В исходном состоянии только часть запросов отдела маркетинга принималась к исполнению, релизы выполнялись раз в полугодие, и лишь 5% рабочего времени сотрудников уходило на разработку новой функциональности. За четыре года удалось перейти к ускорению разработки до 10–15 сборок ежедневно, увеличению продуктивности использования рабочего времени до 40%, снижению времени тестирования с трех недель до одного дня и прочим чудесам.

Приведенный пример при всей своей реальности находится за границей очерченной ранее области «бизнес, имеющий собственный ИТ-отдел». Его можно использовать как поучительную историю, однако с его помощью слож-

[1] https://usa.visa.com/about-visa/our_business/global-presence.html.

[2] http://fortune.com/2017/03/06/apple-iphone-use-worldwide/.

[3] HP LaserJet Firmware Division.

но выделить ограничения применимости DevOps. В то время как список основных трудностей конечен и в целом довольно понятен.

DevOps не очень подходит тем, у кого нет собственной разработки программного обеспечения, например если все 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 не 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 ее 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 два-три 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. Особенности работы в таких ситуациях будут подробнее рассмотрены в разделе «Готовое коммерческое программное обеспечение».

Сложности применения DevOps возникнут и в организациях, где разработка программного обеспечения есть, но программисты выведены за штат: 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 объеме 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 «Необычные команды». Заметим, что если еще пять лет назад было очень модно фокусироваться на основном бизнесе компании, передавая все вспомогательные функции сторонним исполнителям, то в настоящее время наблюдается тенденция к возвращению собственной разработки программного обеспечения и компетенций по эксплуатации информационных технологий обратно в компании. Считается довольно глупым не делать этого, когда конкуренция в совершенно разных отраслях сводится к соревнованию в применении информационных технологий, основанных на программном обеспечении.

> *«Если кратко – программное обеспечение пожирает мир.*
> *Все больше и больше крупных бизнесов в своей основе имеют ПО и используют онлайн-доставку, от индустрии развлечений до сельского хозяйства и даже до вооруженных сил. Многие из лидеров являются технологическими организациями с духом предпринимательства, управляемыми в стиле компаний Силиконовой долины, – они захватывают и полностью перестраивают сложившиеся отраслевые структуры. В ближайшие десять лет еще большее количество отраслей будет разрушено… такими компаниями»*[1].
>
> *Марк Андриссен (Marc Andreessen),*
> *сооснователь Netscape, сооснователь и партнер*
> *венчурного фонда Andreessen-Horowitz, 2011*

Следующее ограничение применения DevOps – устоявшиеся, сложившиеся процессы, подкрепленные 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, приведенные в разделе «Стартап как ориентир», 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, жестко связанная ИТ-архитектура. Организация небольших команд требует возмож-

[1] https://www.wsj.com/articles/SB10001424053111903480904576512250915629460.

ности 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 нее части для отдельных самостоятельных команд, работающих асинхронно, будет достаточно сложно. Некоторые соображения по данному вопросу приведены в разделе «Эволюционирующая архитектура».

К изложенным сложностям следует добавить еще 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 – таких, как описано в разделе «Необычные команды». В некоторых организациях, к примеру, поощряется удаленная работа без необходимости присутствия в офисе в определенные часы. Встречаются территориально распределенные компании, где в том числе и сотрудники ИТ-подразделения не находятся все в одном месте. Наконец, во многих компаниях организационная структура настолько жесткая, что не подразумевает создания кросс-функциональных команд. Все эти примеры иллюстрируют приведенный 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, приведенные в первой части книги (в частности, инфраструктура как программный код, автоматизированное управление конфигурациями), показывают необходимость использования облачных вычислений. Как было показано в разделе «Истоки», весь DevOps стал возможен благодаря им. 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.

Рисунок 5.2 суммирует все основные побудительные мотивы движения в сторону DevOps, а также факторы, ограничивающие применение DevOps:

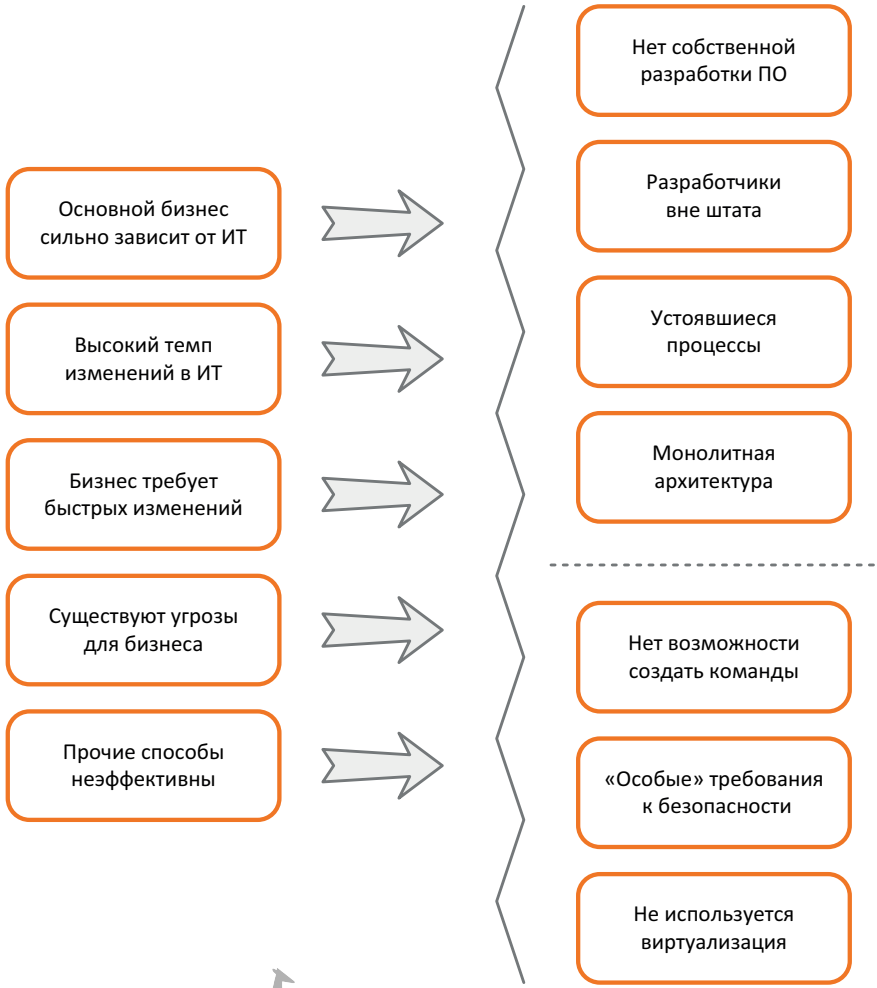

Рис. 5.2 ❖ Интерес к DevOps и известные ограничения

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 все больше и больше. Когда наступает (и наступает ли) тот предел, при котором ограничения складываются в непреодолимый барьер,

неизвестно. Однако пример подразделения компании HP по разработке встроенного программного обеспечения, как и существующие примеры реализации DevOps на архитектуре мейнфреймов, показывает, что, вполне вероятно, граница находится сильно дальше, чем принято думать.

Готовое коммерческое программное обеспечение

Многие организации, стремясь сэкономить на информационных технологиях, уменьшить сложность систем и ускорить получение отдачи, используют принцип «Как можно меньше разрабатывать самостоятельно, как можно больше покупать готового». Класс «коробочного» программного обеспечения даже имеет имя собственное – COTS (англ. Commercial Off-the-Shelf, буквально: коммерческое, с полки). Такой подход имеет веские основания и довольно часто встречается. Однако, как было показано выше, применение готовых программных продуктов вместо собственной разработки ПО является серьезным препятствием на пути применения DevOps. Сталкиваясь с проблемой «совместимости» DevOps и COTS, организации выработали следующий набор рекомендаций.

Во-первых, не следует использовать COTS для автоматизации стратегически важных направлений бизнеса. В условиях, когда соревнование между компаниями-конкурентами смещается в сторону информации и информационных технологий, необходимо иметь максимальные гибкость и контроль, обычно недостижимые при использовании COTS. Поэтому первый совет, который вам даст любой серьезный эксперт: избавляйтесь от готового программного обеспечения, работающего в самых важных для вас областях; переходите на собственную разработку ПО.

Если же применение COTS, пусть временное, является неизбежным, то собственно приложения и стратегию работы с ними следует выбирать в зависимости от следующей условной классификации:

- открытые приложения, такие как Salesforce, позволяют использовать стандартную функциональность, адаптируя ее под свои бизнес-процессы;
- закрытые приложения, такие как продукты компании Adobe, практически не подразумевают никакой адаптации;
- платформенные приложения, такие как Microsoft Dynamics, являются основой для создания ваших собственных ИТ-систем.

С COTS необходимо работать по тем же принципам, что и с собственным программным обеспечением, находящимся в охвате практик DevOps. Следует избавиться от установки и настройки ПО через интерфейс взаимодействия с пользователем или администратором – придется детально изучить процесс установки, понять, что именно делает программа-инсталлятор, какие файлы создаются, какие изменяются, какие базы данных модифицируются и как и т. д. После этого необходимо создать собственный скрипт, повторяющий работу

оригинального инсталлятора[1]. Возможно, потребуется разработать несколько скриптов установки и настройки системы в зависимости от используемых или планируемых сред: тестирования, приемки, среды эксплуатации. Все разработанные скрипты должны быть размещены в системе контроля версий. При необходимости в той же системе либо в системе хранения артефактов следует разместить требуемые для работы приложения библиотеки, двоичные и прочие файлы. По сути, необходимо избавиться от ручного и непонятного процесса установки и настройки приложения, заменив его на автоматизированное развертывание с помощью известных и находящихся под контролем скриптов.

Точно так же следует пересмотреть подход к настройке уже установленного программного обеспечения. Автоматизированные системы контроля позволят выделить те области, которые почти никогда не изменяются при конфигурировании и обновлении приложения. И наоборот, станут видны элементы конфигураций и файлы, изменяющиеся при настройке, а значит, подлежащие тщательному контролю в системе управления версиями. Задача состоит в том, чтобы в любой момент времени иметь отдельно хранимую полную копию всех настроек приложения, находящуюся под версионным контролем.

Как минимум возможны следующие варианты решения данной задачи.

1. В стандартные средства настройки COTS, такие как интегрированная среда разработки (англ. IDE, Integrated Development Environment), встраиваются перехватчики изменений конфигурации системы. Как только администратор или разработчик что-то меняет в приложении, перехватчик замечает сделанные изменения, преобразует их в формат, пригодный для системы контроля версий, и отправляет ей полученный файл или файлы – с возможным контролем конфликтов изменений, сделанных разными сотрудниками. Впоследствии с помощью встроенных механизмов импорта конфигураций можно изменить или восстановить настройки системы уже без использования интегрированной среды разработки. Такой способ контроля конфигураций является наименее затратным, однако поддерживается не всеми ИТ-системами.
2. Выполняется экспорт настроек приложения в формате, пригодном для системы контроля версий. В идеальном случае такой экспорт осуществляется автоматически при наличии триггера изменений в COTS. Если же триггера нет – экспорт выполняется по расписанию с частотой, зависящей от потока изменений в системе (например, каждую ночь). Как правило, системы контроля версий позволяют выполнять сверку поступа-

[1] Показательна ситуация с регулярным созданием так называемых портативных версий (англ. Portable) некоторых популярных прикладных программ силами энтузиастов, без какого-либо доступа к исходному коду и без знания внутреннего устройства этих приложений. Такие версии могут быть запущены и полноценно использоваться без инсталляции на целевой компьютер. Раз такую работу, в том числе для довольно сложных приложений, могут выполнить энтузиасты – значит, можно рассмотреть возможности и для промышленных систем.

ющих файлов с хранимыми, и в случае отсутствия изменений нагрузка при отслеживании будет незначительной, а информация не будет дублироваться. Такой способ является более дорогим, по сравнению с первым, однако он более универсален.

3. Самый затратный способ для случаев, когда COTS не позволяет выполнять экспорт конфигураций, но имеет возможности импорта, заключается в разработке собственного приложения для настройки COTS. Вся работа по конфигурированию выполняется в таком приложении, файлы настроек в требуемом формате размещаются в системе контроля версий и в формате COTS импортируются в целевую ИТ-систему.

По сути, речь идет о воссоздании части среды разработки для приложения, исходный код которого недоступен. Однако на этом работа не завершается. Следует разработать тесты, пригодные для автоматического выполнения, проверяющие корректность вносимых изменений, работоспособность системы, интеграции и взаимодействие с другими системами. Автоматизированное тестирование – часть конвейера развертывания, а конвейер должен быть построен и для COTS в том числе.

Программой-максимум для COTS является регулярное быстрое автоматическое пересоздание приложения в среде эксплуатации с нуля на основе данных системы управления конфигурациями, без простоя системы и незаметно для пользователей. Достижение такого уровня контроля обеспечит отсутствие сюрпризов при изменениях «коробочной» системы и скорейший, в том числе автоматический, откат неудачных изменений.

Эволюционирующая архитектура

Ранее в разделе «Область применения и ограничения DevOps» была обозначена сложность, имеющаяся у большого числа организаций, образованных до 2010-х годов, – монолитная ИТ-архитектура с тесными связями между компонентами систем (рис. 5.3). Любое современное приложение состоит из множества объектов, взаимодействующих между собой: взаимосвязанные структуры создаются на уровне бизнес-логики, данных, ИТ-инфраструктуры и др. Приложение проектируется и разрабатывается как единое целое, как горизонтально (объекты и связи между ними), так и вертикально (приложения, серверы, СУБД, протоколы и интерфейсы обмена данными). С такой архитектурой связан целый набор сложностей:

- даже небольшие изменения в одной части системы могут привести к негативным, зачастую непредсказуемым последствиям в другой части;
- над функциональностью системы одновременно трудится много сотрудников, каждый в своей части, что требует ресурсов на координацию отдельных исполнителей;
- очень немногие сотрудники знают, как в целом устроена ИТ-система, что от чего зависит, что допустимо, а что нет; такие сотрудники быстро становятся очень ценными, незаменимыми и перегруженными;

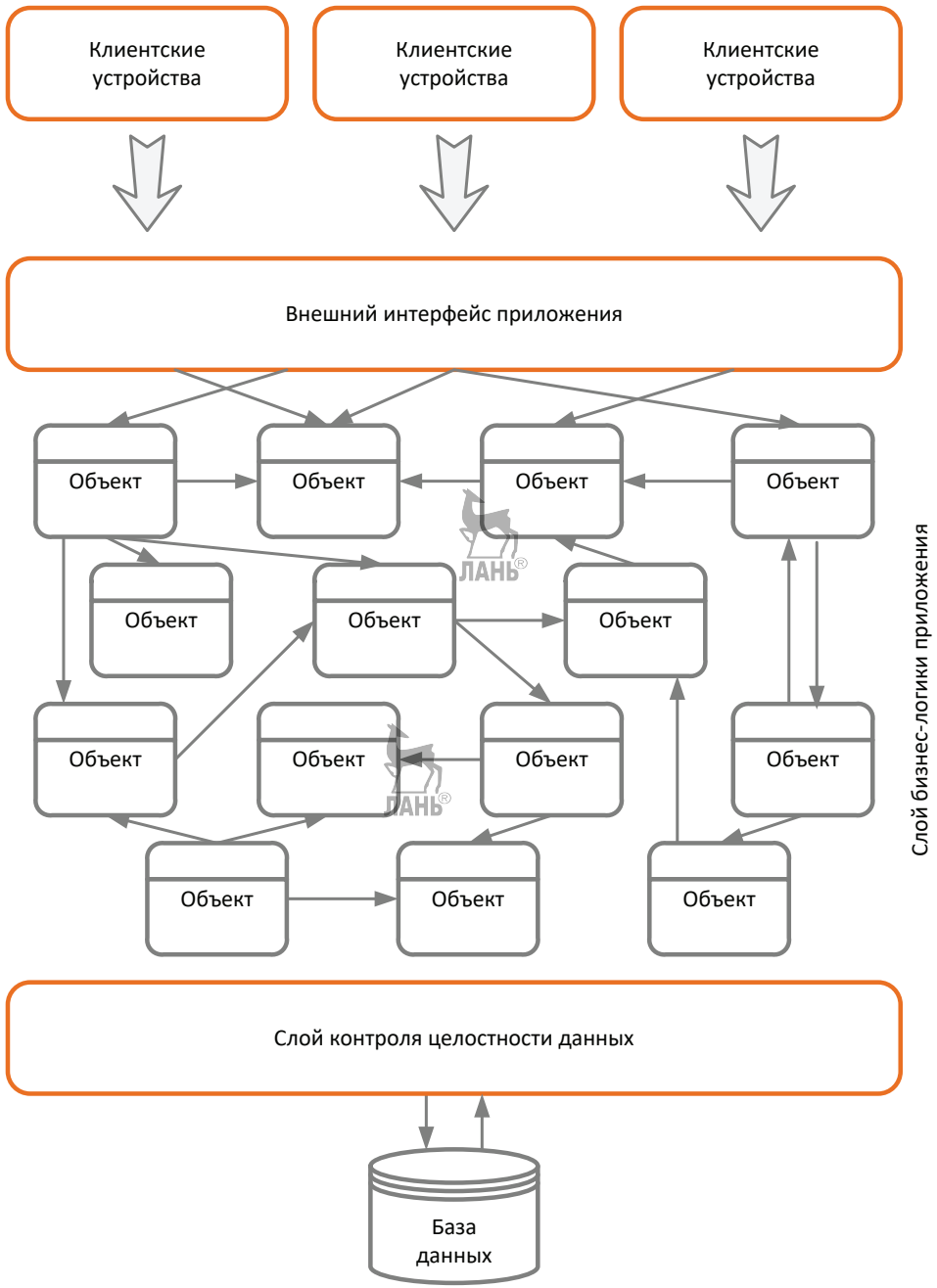

Рис. 5.3 ❖ Пример монолитной архитектуры простой ИТ-системы (показаны не все связи)

- любая создаваемая документация по ИТ-системе довольно быстро устаревает;
- разработка и эксплуатация ИТ-системы разделены естественным образом: сотрудники сопровождения и поддержки не погружаются в сложные подробности внутреннего устройства приложения, а потому даже по относительно простым вопросам вынуждены обращаться к разработчикам – теряя время на ожидание и отвлекая программистов, архитекторов и бизнес-аналитиков от других задач;
- сложно выделить отдельные области для небольших самодостаточных команд, поэтому основные преимущества гибкой разработки сводятся на нет и не дают ожидаемого эффекта;
- имеющаяся архитектура не отвечает в полной мере требованиям текущего момента времени, она вскоре после создания становится устаревшей – ключевые решения принимались тогда, когда информации было меньше, опыта разработки и использования приложения также не было, а жизнь еще не успела внести свои коррективы;
- изменять и развивать саму архитектуру непросто ввиду большого числа жестких связей.

Ответом на данные сложности традиционно является выстраивание формальных процессов управления и увеличение числа точек контроля, информирования и согласования, замедляющих внесение изменений. К примеру, корректировка одной строки программного кода для устранения выявленного дефекта может занять несколько минут у разработчика, но несколько месяцев до появления в среде эксплуатации. Приходится выполнять тестирование всей системы целиком, объединяя изменения, подготовленные множеством исполнителей. Поэтому ИТ-подразделения стремятся ввести дополнительные искусственные барьеры в виде календаря релизов: действительно, если тестирование изменений требует больших ресурсов и выполняется сложно, то и затевать такую работу следует как можно реже. К сожалению, даже такая технология работы не защищает в должной мере от появления дефектов в продуктивной среде.

Перечисленные выше проблемы, связанные с монолитной архитектурой, проявились довольно давно, как только был получен первый опыт эксплуатации и развития больших информационных систем. Инженерная мысль находится в постоянном поиске лучших вариантов: были придуманы модульные архитектуры, архитектура вида микроядро, архитектура, управляемая событиями (с использованием брокеров или медиаторов), сервис-ориентированная архитектура (англ. SOA, Service-Oriented Architecture) и др., включая гибридные на основе перечисленных. Однако, как отмечают авторы книги «Построение эволюционирующей архитектуры»[1], все они обладают достаточно существен-

[1] *Ford N., Parsons R., Kua P.* Building Evolutionary Architectures. 2017. ISBN 978-1-491-98636-3.

ными недостатками при зачастую не меньшем усложнении применяемых решений, что делает их слабо применимыми для DevOps-инициатив.

Радикальным многообещающим ответом, получившим большое внимание в последние годы, является так называемая микросервисная архитектура (рис. 5.4). В ней приложение проектируется как набор элементов, выделенных по принципу доменов: каждый из них «отвечает» за определенную сущность ИТ-системы и включает в себя все необходимые технические и инфраструктурные компоненты, от которых зависят, к примеру, базы данных и библиотеки. Такие сервисы не связываются между собой – они «общаются» исключительно через заданные программные интерфейсы или очереди сообщений. Ни один из сервисов не должен знать о внутреннем устройстве других сервисов и не должен иметь с ним никаких зависимостей, используется принцип «ничего общего» (англ. Share Nothing).

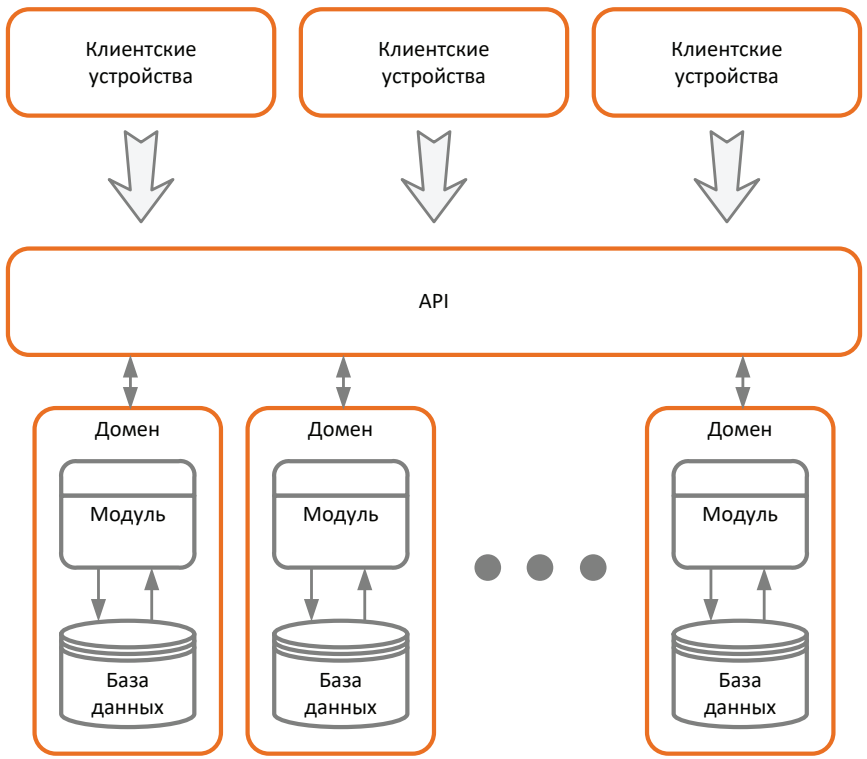

Рис. 5.4 ❖ Пример микросервисной архитектуры простой ИТ-системы (показаны не все домены)

Следование приведенным рекомендациям позволяет добиться ситуации, когда модификацию каждого из сервисов можно выполнять независимо от других – силами отдельной команды. Работая как с каждым из сервисов от-

дельно, так и с ИТ-системой целиком, можно опираться на все базовые принципы DevOps: поток создания ценности, конвейер развертывания, хранение всего в системе контроля версий, автоматизированное управление конфигурациями, определение завершения. Календари релизов и связанные с ними многомесячные ожидания более не требуются, а процесс управления изменениями может быть сильно упрощен.

Не менее существенной является возможность перехода к эволюционирующей архитектуре: постоянно развивающейся вслед за новыми требованиями бизнеса и не отстающей от появляющихся ценных технологических новинок. Например, команда, отвечающая за домен А, может подготовить следующую версию функциональности, не отключая текущую. Сервис В, использующий старую версию сервиса А, будет продолжать это делать без потери качества. В то же время сервис С, рассчитанный на новую версию сервиса А, сможет обращаться к нему за новой функциональностью. Постепенно все приложение будет переведено на использование улучшенной версии для данного домена А, и предыдущую версию можно будет отключить – при этом не потребуется ожидания готовности всех компонент к единовременной и масштабной миграции. Точно тем же способом, независимо от других команд и доменов, можно выполнять рефакторинг отдельных сервисов, снижая накопленный технический долг.

Безусловно, использование микросервисной архитектуры влечет за собой ряд сложностей. Выделение доменов требует серьезной проработки и вряд ли может быть сделано раз и навсегда – пересмотр структуры сервисов ИТ-системы должен быть постоянной деятельностью. Необходимо следовать четким правилам определения и документирования программных интерфейсов и версионности. Обеспечение ссылочной целостности данных в значимой степени смещается с уровня системы управления базами данных на уровень домена. Потребуется организация мониторинга для каждого из сервисов: не только для контроля работоспособности, но и для отслеживания использования.

В существенной мере распространению микросервисных архитектур в последние годы способствовало появление технологии контейнеризации. В ней для организации изолированного рабочего пространства для какого-либо сервиса не требуется выделять отдельную полную виртуальную машину, а все операции по созданию и управлению контейнерами могут быть выполнены программным способом, включая автоматическое динамическое добавление дополнительных мощностей при повышении спроса и такое же автоматическое освобождение ресурсов при снижении.

Однако для большинства организаций переход на микросервисы отнюдь не прост. Некоторые компании выделяют финансирование и запускают большие проекты, направленные на «переписывание» имеющихся информационных систем в новой архитектуре. Такие проекты занимают месяцы и годы, не приводя, как правило, к успеху, что объясняется двумя причинами. Во-первых, масштабность преобразований настолько высока, что довести работу до конца в разумные сроки не представляется возможным, даже если остановить все прочие инициативы. Во-вторых, одновременно с проектом по новой разработке имеющаяся

система продолжает развиваться в ответ на потребности бизнеса, что делает сложно реализуемой синхронизацию функциональности обоих приложений.

Вместо запуска большого проекта, имеющего мало шансов на успех (и совершенно определенно не предлагающего «быстрых побед»), эксперты рекомендуют[1] выстраивать модернизацию архитектуры как постоянную деятельность, являющуюся частью обычной работы по разработке и развитию. Анализируя очередной бизнес-запрос, можно попробовать выделить часть имеющейся системы в отдельный домен, для которого сразу же создать необходимое окружение: программный интерфейс взаимодействия с основной системой, соответствующий набор тестов, конвейер развертывания. Постепенно, шаг за шагом, отдельные части монолитной системы будут реализовываться в микросервисной архитектуре, при этом основным драйвером изменений будут бизнес-потребности (рис. 5.5).

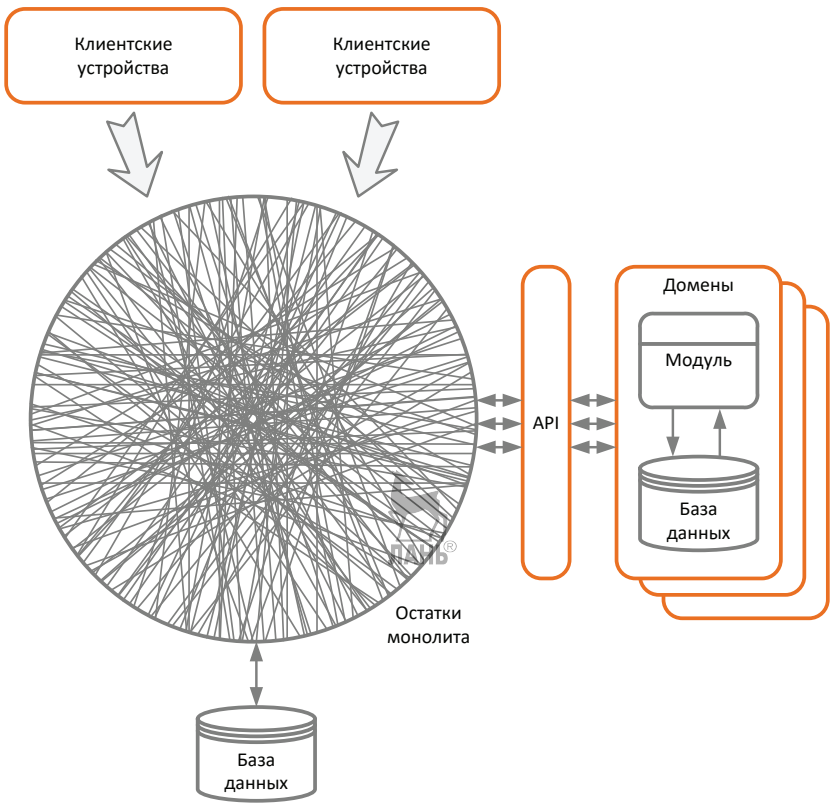

Рис. 5.5 ❖ Вариант постепенного перехода к микросервисной архитектуре для имеющегося приложения

[1] *Humble J., Molesky J., O'Reilly B.* Lean Enterprise: How High Performance Organizations Innovate at Scale. 2015. ISBN 978-1449368425.

Совместимость с сервисным подходом

> Миф: DevOps несовместим с ITIL.
> DevOps-практики могут быть сделаны совместимыми с процессами ITIL. Однако, чтобы обеспечить быстрое время выпуска и высокую частоту релизов, многие части ITIL-процессов станут полностью автоматизированными, решая множество проблем с процессами управления конфигурациями и релизами. А поскольку DevOps требует быстрого обнаружения и устранения сервисных инцидентов, дисциплины ITIL, связанные с проектированием услуг, управлением инцидентами и проблемами, остаются такими же востребованными, как и прежде.
>
> *Цитата из книги «Руководство DevOps»*[1]

За последние двадцать лет многие компании инвестировали в ITIL миллионы фунтов стерлингов, долларов и евро. Траты вполне обоснованы: организации ищут решения управленческих проблем, связанных с использованием информационных технологий, борются за повышение эффективности ИТ-департаментов. В области управления корпоративными ИТ такие своды знаний, как ITIL и COBIT, являются общепризнанными отраслевыми стандартами, хотя называть их именно стандартами не совсем корректно. Компании ожидали пользы от ITIL, многие компании подобную пользу получили и получают.

Однако такие главы книги, как «Принципы» и «Основные практики», содержат большое количество тезисов и примеров, которые будет нелегко принять обычному, традиционному ИТ-отделу, – уж слишком сильно новые практики расходятся с теми, что применяются сейчас в подавляющем большинстве крупных компаний. Нет ли в этом проблемы?

Эксперты по DevOps считают, что принципиальных сложностей нет. Цитата из книги «Руководство DevOps», приведенная в начале раздела, является типичным примером. Эксперты со стороны управления ИТ-услугами еще более воодушевлены новыми перспективами и в срочном порядке осваивают новую религию. Редкая конференция по ITSM в 2017 году прошла без докладов, а то и целых секций, посвященных DevOps и цифровой трансформации. Можно сказать, что почти все эксперты единодушно считают, что выстроенные по ITIL процессы управления можно каким-либо образом доработать или даже «совместить» с DevOps, сохранив инвестиции. Однако не все так просто.

Между идеями DevOps и идеями ITSM существует фундаментальное противоречие, требующее разрешения (рис. 5.6). Один из двух ключевых постулатов

---

[1] *Kim G., Humble J., Debois P., Willis J.* The Devops Handbook: How To Create Worldclass Agility, Reliability And Security In Technology Organizations. 2016. ISBN 978-1-942-78800-3.

управления ИТ-услугами (помимо организации деятельности в виде процессов) – это представление ценности от информационных технологий в виде услуг. Неотъемлемой частью предоставления услуг являются заказчик и поставщик: первый определяет, что и зачем необходимо получить, второй берет на себя риски и затраты, связанные с услугами. Договоренности при этом максимально подробно формулируются в сервисном соглашении (англ. SLA, Service Level Agreement), закрепляющем ответственность сторон. В случае если заказчик недоволен качеством получаемых услуг, он может попытаться повлиять на поставщика посредством подписанного контракта либо даже сменить одного поставщика на другого. В случае если поставщик посчитает, что данный клиент вызывает больше сложностей, чем прибыли, он может расторгнуть договор и сфокусироваться на предоставлении услуг другим заказчикам. Разумеется, для поставщиков, целиком находящихся внутри какой-либо корпорации на позиции внутреннего ИТ-департамента, ситуация не такая простая, но базовый принцип остается тем же самым.

В то же время DevOps в самой существенной степени опирается на понятие единой команды, в которую входят не только сотрудники ИТ, но и представители бизнеса. Работая совместно друг с другом, они больше фокусируются не на краткосрочных результатах и уж тем более не на соблюдении формальных договоренностей вроде SLA, а на долгосрочных победах. Они вместе идут по дороге, которая становится видна лишь по мере движения. Они договорились, что в случае неудач не ищут виноватых, а извлекают уроки и учатся на собственных ошибках. В предельном случае граница между ИТ и бизнесом исчезает вовсе, что совершенно не похоже на описанный выше подход «мы и они».

Рис. 5.6 ❖ Суть ITSM и суть DevOps

Ответ на данное большое противоречие еще только предстоит найти, однако есть расхождения и, условно говоря, в мелочах, например:

- как уже отмечалось выше, практики DevOps во многом расходятся с привычным порядком выполнения работ в традиционном ИТ-отделе – настолько, что многие ИТ-руководители в корне не готовы принять новые идеи;
- финансирование организуется совершенно иначе: средства выделяются не на проекты, а на продукты;
- многие годы компании работали с ИТ-департаментом по принципу «оптимизация расходов», теперь же предлагается перейти на принцип «оптимизация скорости»;
- управление изменениями согласно ITIL призвано уменьшать риски внесения дефектов и создания простоя за счет относительно медленного и строго формализованного процесса со множеством контролей, уведомлений, согласований и утверждений. Изменения «по DevOps» следует выполнять максимально быстро с должным тестированием, отмечая следы изменений автоматически в соответствующих системах;
- управление конфигурациями и создание конфигурационной базы данных CMDB (англ. Configuration Management Database), описанные в ITIL, редко встречаются в реальной жизни из-за чрезмерной трудоемкости и обилия ручных операций по накоплению и актуализации информации. В то же время управление конфигурациями в DevOps выполняется в существенной степени автоматизированно и в обязательном порядке, настолько, что собственно термин «конфигурации» меняет свой смысл;
- понятие «релиз» изменяется от «релиз – совокупность изменений, подготавливаемых, тестируемых и выполняемых одновременно» до «новая функциональность, доступная клиентам»;
- принципы работы управления инцидентами, включающие разделение на линии поддержки и функциональную эскалацию, заменяются другим принципом: «Вы это разработали, вам и эксплуатировать»;
- управление проблемами (корневыми причинами инцидентов) так и не обретает смысл, его сложно организовать в ITSM и не очень нужно – в DevOps;
- управление мощностями по ITIL работает на основе плана мощностей, в котором должны быть предусмотрены все потребности в ИТ-ресурсах и который привязан к циклу бюджетирования компании, обычно ежегодному. В DevOps мощности должны быть доступны в тот момент, когда они требуются, без потерь времени на поиск поставщика, заключение контракта, ожидание поступления на склад и прочего.

И т. д. Получается, куда ни посмотри – везде рекомендации ITIL отличаются от идей и практик DevOps. Возможно, эти расхождения незначительны и потребуется лишь легкая подстройка имеющихся ITSM-процессов, однако более вероятен сценарий, при котором процессы ITIL придется изменить до неузнаваемости.

Культ карго

Огромное количество команд, стремящихся освоить новые управленческие инструменты, не придает 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 заведем у себя канбан. Конвейер DevOps без автоматизации не бывает? Что ж, поручим ребятам выбрать и настроить несколько систем. И т. д.

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. Ученые, 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 определенные действия и обряды, как правило, под руководством шамана или старейшины. Примеры и подтверждения культа карго были обнаружены и в более ранние времена, самым давним документированным примером является культ на островах Фиджи в 1885 году. Говорят, некоторые проявления такого культа сохранились в отдельных частях Океании до наших времен.

Самым ярким и наиболее известным примером является история, произошедшая во время и непосредственно после Второй мировой войны на островах Меланезии, которые получили стратегически важное значение для боевых действий. Сначала на островах высадились японские вооруженные силы, привезя с собой невиданные ранее товары, одежду, лекарства, запасы пищи, оружия и прочего. Затем острова перешли под контроль антигитлеровской коалиции – так или иначе, но местное население прочно связало появление полезных товаров, которые невозможно произвести, с прилетающими откуда-то белыми людьми. Вскоре после окончания войны острова утратили свою важность, военные базы были свернуты, иностранцы покинули территорию. Аборигены в поисках способа восстановить поток товаров занялись максимально точным воспроизведением всех тех условий, при которых буквально с неба сыпались богатства, а именно: они стали разукрашивать себя в цвета, повторяющие форму американской армии, маршировать по плацу так, как это делали военные, использовать копии ружей, сделанные из бамбука, максимально полно копировать все другие известные им внешние проявления. Дошло до построения зданий, копирующих командные пункты аэродромов, включая внутреннее оборудование и антенны – сделанные, однако, все так же из бамбука (рис. 5.7). Были расчищены дополнительные площадки в джунглях, чтобы создать больше «аэродромов», на них установлены довольно точные ко-

пии самолетов. Разумеется, все эти действия не привели к возвращению чужеземцев, равно как и к новым поступлениям товаров.

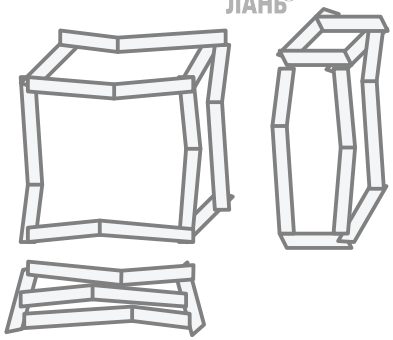

Рис. 5.7 ❖ Современный бамбуковый компьютер, допускающий виртуализацию

Любому образованному человеку ясно, что эта история не могла закончиться иначе – для получения полезных продуктов недостаточно копировать чужую деятельность. Однако то, что очевидно в примере с аборигенами, зачастую совершенно не понятно в ситуациях с собственными сотрудниками и коллегами. Бездумное воспроизведение ритуалов гибкой разработки программного обеспечения в надежде ускорить вывод продуктов на рынок встречается в практике различных компаний чаще, чем следует.

Примечательно, что при становлении управления ИТ-услугами явление культа карго наблюдается не менее часто. К сожалению, достаточной информации для анализа закономерностей пока нет.

Начинать с малого, действовать сегодня

— Что бы вы рекомендовали ИТ-менеджерам делать с DevOps?
— Начинайте прямо сейчас! Сегодня – самый лучший день для старта.

Интервью Гэри Грувера (Gary Gruver), руководителя, навсегда изменившего HP LaserJet Firmware Division[1]*, 2017*

Предыдущие разделы книги должны были создать у читателя ощущение масштабности перемен – использование DevOps подразумевает иные принципы управления, иные базовые подходы к использованию информационных технологий. Многие знают из собственного опыта, что большие преобразова-

[1] https://cleverics.ru/subject-field/interviews/723-gary-gruver-interview-on-devops.

ния никогда не даются легко. Даже при наличии серьезных сложностей с ИТ-инфраструктурой, неудобных вопросов от основного бизнеса относительно скорости реализации изменений, давления со всех сторон – все равно руководители будут стремиться оставаться в зоне относительного комфорта, где все более-менее понятно и предсказуемо, пусть и не так красиво выглядит, как хотелось бы. Большой размер необходимых изменений не должен смущать и останавливать, вопрос ни в коем случае не стоит как «Все или ничего!».

Если вы находитесь в ситуации, совпадающей с критериями раздела «Область применения и ограничения DevOps», то ждать действительно нечего. Находится ли движение DevOps в начале своего пути? Безусловно: остается еще много вопросов, особенно в области корпоративных информационных технологий. Стоит ли дожидаться, когда кто-то найдет все ответы? Ни в коем случае! Как было изложено в самом начале книги, первопроходцы, возможно, идут не оптимальным путем, зато накапливают свой собственный опыт, позволяющий им двигаться к цели быстрее догоняющих. В настоящее время литературы, мероприятий и апологетов DevOps столько, что информационный вакуум просто невозможен. Напротив, все острее стоит задача фильтрации информации, ведь, как известно, большая часть написанного в интернете не соответствует действительности, маркетинговый шум скрывает реальные сложности и неудачи, а «экспертом» внезапно для окружающих теперь может стать каждый. Тем важнее составить свое собственное представление о предметной области, поставить свои вопросы и найти на них ответы.

Как и с управлением ИТ-услугами, частая ошибка – попытка «внедрить» DevOps. Разумеется, внедрить DevOps нельзя, это словосочетание имеет не больше смысла, чем «внедрить здоровое питание» или «внедрить молоток». DevOps можно применять, использовать – как можно использовать другие управленческие инструменты, направленные на решение конкретных задач организации. DevOps – не программный продукт, который можно инсталлировать и запустить, и не инженер, которого можно нанять и которому можно поручить навести порядок в ИТ. DevOps во многом требует культурных и организационных изменений, и не только в департаменте информационных технологий.

Бизнес-подразделения также должны будут измениться, ровно в тех же ключевых направлениях, что и ИТ-отдел: организационно, культурно и инструментально. Наивно полагать, что по иным принципам работающий ИТ-департамент может тем же привычным способом взаимодействовать с бизнес-подразделениями, как это было всегда. Напротив, идеальная модель DevOps подразумевает исчезновение не только границы между разработкой и эксплуатацией, но и между ИТ и бизнесом. Если же до идеала пока далеко, как минимум потребуются иные способы взаимодействия и новые принципы финансирования информационных технологий. Такого рода изменения невозможны без мощной и безусловной поддержки на самом высоком уровне бизнеса, равно как и на высшем уровне ИТ компании. «*Вы можете рассчитывать на мою всестороннюю поддержку, кроме ресурсов, бюджета, полномочий и распоряжений, и чтобы мне не пришлось ничего делать*», – такой вариант, безусловно, не годится.

К применению DevOps ни в коем случае нельзя подходить, как к проекту. Проектная организация работ подразумевает получение уникального результата в ограниченные сроки и в рамках заданного бюджета, в то время как DevOps означает игру в долгую – от сегодня и навсегда. Таким образом, нет более бессмысленного словосочетания, чем «Проект внедрения DevOps».

Необходимо обучать персонал. Не обучить, а именно обучать; более того, необходимо создать условия и механизмы обмена знаниями, полученным опытом. Такие механизмы следует постоянно тестировать. Те из них, которые прижились в компании и используются специалистами, следует поощрять. Неработающие – убирать, взамен придумывая новые. Нельзя допускать перекосов, например в изучении исключительно технических аспектов построения конвейера развертывания. За деревьями можно не разглядеть лес, а построенный в одной части организации конвейер не даст ощутимых преимуществ. Следует уделять соизмеримое внимание обучению персонала идеологическим аспектам DevOps, созданию новой культуры организации и выполнения работ в ИТ.

Как и с другими организационными преобразованиями, персонал, скорее всего, разделится на группы: некоторые будут приветствовать изменения и прикладывать дополнительные усилия, другие будут относиться нейтрально, больше присматриваясь, третьи же будут активно против либо саботировать движение вперед. В этом отношении DevOps ничем не отличается от других организационных изменений, для выполнения которых менеджерами накоплен достаточный багаж инструментов.

Нередко встречающийся подход в компаниях с уже имеющейся ИТ-инфраструктурой и системами – определить и выделить часть систем, слабо связанную с другими системами и находящуюся, как правило, в области современных цифровых приложений. В этой сфере, как правило, проще запустить базовые элементы DevOps, включая поток создания ценности, конвейер развертывания, систему контроля версий, автоматизированное управление конфигурациями и т. д. Накопленный опыт можно затем попробовать транслировать на другие области, однако не следует ожидать, что это будет легко. К сожалению, различные ИТ-системы обладают своими особенностями, равно как и ИТ-команды, и бизнес-подразделения. Тем не менее, начиная с более простых случаев, можно более уверенно двигаться дальше.

Объяснить себе и окружающим, почему не следует заниматься новой темой или почему новые практики не заработают и не приживутся, довольно легко. Это известная когнитивная ловушка, которую можно обойти только действиями.

> Келси Хайтауэр (Kelsey Hightower), адвокат по развитию персонала подразделения Google Cloud Platform, высказывается довольно категорично:
>
> «*Нечего даже обсуждать, ставки сделаны на непрерывное развертывание и DevOps. Я считаю: послушайте, разберитесь уже с этим, или идите работать в другую компанию, где вы сможете разобраться*»[1].

[1] http://www.zdnet.com/article/time-to-move-on-from-devops-and-continuous-delivery-says-google-executive/.

Поток создания ценности как основа

Предположим, небольшая и ограниченная область определена. Есть намерения построить использование и управление информационными технологиями в ней так, чтобы получить больше пользы: быстрый выпуск новых продуктов, оперативное тестирование бизнес-идей, антихрупкость и управляемый технический долг. Что конкретно следует делать в первую очередь?

Те, кто уже ходил по этой дороге, советуют начинать с создания команды. Чем ближе такая группа людей будет к описанной в разделе «Необычные команды», тем выше шансы на успех.

Затем следует выполнить картирование потока создания ценности в состоянии «как есть». Данное упражнение поможет создать общее, единое понимание текущего процесса, а затем провести анализ узких мест и поиск потерь. Еще до каких-либо следующих шагов можно попытаться изменить поток так, чтобы потери стали меньше. Также можно сделать список гипотез, требующих проверки, о местах, задержках и действиях, вносящих максимальные потери. Этот список пригодится позже, он станет основой для последующих улучшений.

После этого можно переходить к построению конвейера развертывания для той части потока, которую можно перевести в автоматизированное русло. Увлекаться тотальной автоматизацией не следует, на первых порах достаточно базового конвейера, выполняющего хотя бы сборку и первичное тестирование. Первый же опыт использования конвейера подскажет, в каком направлении его наиболее целесообразно развивать. Помня об ограниченных ресурсах, не следует сразу же ставить себе объемные задачи и стремиться к некоему недостижимому идеалу.

Намного важнее встроить в поток создания ценности механизмы измерения ключевых показателей. Для каждой точки потока их можно придумать множество, однако задачи максимизации числа метрик не стоит. Первое время можно обойтись тремя самыми важными: время выпуска, время обработки и доля работ, выполненных без ошибок. Постоянный контроль этих основных показателей подскажет места, где улучшения дадут наиболее значимый эффект.

Поняв и отыскав области для совершенствования, можно разработать следующую версию потока, «как надо», затем подготовить список изменений и запланировать их выполнение. К изменениям, связанным с улучшениями деятельности, следует относиться не как к разовым мероприятиям, а как к постоянной работе: отныне в данной области принято регулярно, активно и методично искать места возникновения потерь и устранять их. Это – ежедневная задача каждого в команде.

Для устранения узких мест и минимизации потерь можно подбирать те инструменты из арсенала DevOps, бережливого производства и гибкой разработки программного обеспечения, которые наилучшим образом решат сформулированную задачу. Таким образом, не книжные практики определяют действия персонала, а анализ потока создания ценности ставит цели и помогает подобрать наиболее подходящие инструменты.

Затем цикл замыкается: после реализации запланированных изменений необходимо понять, произошли ли ожидаемые улучшения, каковы теперь значения ключевых показателей, где находится следующее узкое место и что можно предпринять для его устранения. Как было многократно сказано, главное – выйти на эту дорогу и начать движение по ней, а конечной точки у нее нет.

Некоторые команды приводят пример, когда целевое, конечное состояние потока «как надо» разрабатывается на раннем этапе, а затем начинается планомерное и постепенное движение по переходу в это состояние. Такой путь представляется более сложным и рискованным, так как вероятность ошибиться на раннем этапе велика. Более того, необходимости в проработке конечной точки может и не быть, а ее достижение возможно путем постоянного экспериментирования.

Построив поток создания ценности в отдельной небольшой области, следующим логичным шагом будет перенос опыта и практик на другие области – об этом уже было написано ранее. Такой подход возможен, однако самые интересные и сложные задачи начинаются тогда, когда несколько отдельно работающих DevOps-команд необходимо объединить в нечто большее либо когда в DevOps-движение данной компании вовлечено несколько десятков человек. Однако вопрос масштабирования DevOps на крупные организации – отдельная и большая область знаний, выходящая за рамки данной книги.

Краткое резюме главы

Беспристрастное рассмотрение какой-либо темы требует объективного, насколько это возможно, изложения сути, особенностей, преимуществ и ограничений. Наименее желательна ситуация, при которой у читателя сложится искаженное представление о DevOps как о наилучшем средстве решения всех вопросов управления современными информационными технологиями. Глава о вопросах применения DevOps получилась неожиданно объемной, при этом обозначенные нюансы и сложности зачастую масштабны, а ответы на неудобные вопросы – что делать с готовым программным обеспечением, монолитной архитектурой, сервисным подходом – неочевидны.

Зато очевидно другое: либо придется самостоятельно искать на них ответы, либо ждать, пока кто-то расскажет о своих достижениях. В таком выборе ключевое слово – «своих». Чтобы достижения стали своими, необходимо действовать, а не ждать.

Заключение

DevOps имеет свои истоки, предпосылки появления. Для возникновения DevOps-движения к 2010-м годам сложились определенные 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 чем-то циничный расчет.

Еще один-два года назад можно было бы спорить о том, что такое DevOps, что входит в это понятие, что находится за границей, зачем все 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 развертывания) все более-менее понятно, то ключевая проблема – как получить управленческую пользу от DevOps в традиционных компаниях?

Представляется, что ближайшие годы будут посвящены корпоративному DevOps. Нам предстоит многое узнать, многому научиться.

Вполне возможно, что читатель, закрывая эту книгу, подумает: «*Хорошая книга, все понятно, но я и так все это знал*». Это будет лучшим комплиментом автору – структурированное, логичное, убедительное и по возможности беспристрастное изложение довольно непростой темы принесло свои плоды.

Приложение

Тест «Есть ли у вас DevOps»

Для каждой устоявшейся области менеджмента со временем появляется линейка, зачастую называемая уровнем зрелости. Отдельные компании по заранее определенным критериям могут попробовать оценить наличие у себя различных рекомендуемых практик, чтобы затем вывести интегральную оценку по условной шкале – насколько те или иные аспекты управления хороши или плохи в данной конкретной организации. Наиболее часто такие уровни зрелости применяют всевозможные консультанты, проводя так называемые обследования, диагностики или аудиты. Забавно, что наименее зрелые консультанты наиболее часто используют уровни зрелости (простите за каламбур), рекомендуя своим несчастным клиентам *«скорее подтянуть уровень всех процессов хотя бы до троечки по пятибалльной шкале»*. В противоположность описанным реалиям есть мнение других экспертов о том, что уровни зрелости – в целом штука малополезная, так как в лучшем случае отвечает на вопрос «сколько есть», но не «сколько нужно», а потому не дает ценной информации для принятия решений.

Тем не менее многим очень хочется понять – хорошо ли мы работаем? Лучше ли других? Что сказали бы гуру? Отстаем или обгоняем других? К счастью, общепринятых моделей уровней зрелости для DevOps пока не появилось. Поэтому имеющийся спрос можно удовлетворить нижеследующим почти бессмысленным тестом, который призван показать, насколько ваша ИТ-организация близка к условной идеальной DevOps-команде.

Постарайтесь по возможности честно ответить на каждый из вопросов выбором одного из предложенных вариантов, суммируя полученные баллы.

1. Мы регулярно измеряем время выпуска (Lead Time), время обработки (Process Time) и долю работ, выполненных без ошибок (%C/A):
   a) для всех изменений – 5 баллов;
   b) время от времени – 3 балла;
   c) именно это не измеряем, зато измеряем много другого – 1 балл;
   d) измерения суть зло и провокация – 0 баллов.

2. Наше время выпуска в среднем составляет:
   a) несколько часов – 5 баллов;
   b) несколько дней – 3 балла;
   c) несколько недель – 1 балл;
   d) несколько месяцев – 0 баллов.

3. Наша частота релизов в среду эксплуатации составляет:
   a) несколько раз в день – 5 баллов;
   b) несколько раз в неделю – 3 балла;
   c) несколько раз в месяц – 1 балл;
   d) у нас график релизов по кварталам и дольше – 0 баллов.

4. Мы выпускаем релизы:
   a) мы не выпускаем, бизнес выпускает – 5 баллов;
   b) так часто, как того требует бизнес – 3 балла;
   c) обычно редко, но можем и вне очереди – 1 балл;
   d) в соответствии с политикой релизов, когда накапливаются изменения – 0 баллов.

5. Простой при обновлении среды эксплуатации составляет:
   a) ничего не составляет – 5 баллов;
   b) несколько минут – 3 балла;
   c) несколько часов – 1 балл;
   d) у нас есть согласованное с бизнесом время отключения систем для обновления – 0 баллов.

6. Мы вносим разрушения в среду эксплуатации:
   a) мы не вносим, это постоянно делают специально разработанные скрипты и системы – 5 баллов;
   b) при тестировании и развертывании – 3 балла;
   c) почти каждый день, когда выполняем обычную работу – 1 балл;
   d) у нас разрушения невозможны, все стабильно – 0 баллов.

7. Мы выделяем время на улучшения и инновации:
   a) до 20% рабочего времени – 5 баллов;
   b) не регулярно, но стараемся – 3 балла;
   c) мы улучшаемся, выполняя обычную работу – 1 балл;
   d) нас улучшают специальные люди – 0 баллов.

8. Наш бизнес ставит эксперименты на живых пользователях:
   a) каждый день – 5 баллов;
   b) время от времени – 3 балла;
   c) когда ИТ-отдел разрешает – 1 балл;
   d) эксперименты не нужны, так как аналитики про пользователей все и так прекрасно знают – 0 баллов.

9. Наш конвейер развертывания:
   a) работает полностью автоматически – 5 баллов;
   b) имеет несколько ручных шагов – 3 балла;

c) у нас нет конвейера – 1 балл;
d) в наших условиях конвейер невозможен – 0 баллов.

10. Мы приоритизируем задачи в потоке создания ценности:
    a) на основе стоимости задержки – 5 баллов;
    b) быстро прикидывая выгоду, ресурсы и срочность – 3 балла;
    c) мы играем в покер – 1 балл;
    d) мы не приоритизируем, это делают за нас – 0 баллов.

11. Мы выпускаем минимальный рабочий продукт (MVP), чтобы:
    a) наименьшими ресурсами получить максимально полную информацию для принятия решения – 5 баллов;
    b) понять, стоит ли двигаться дальше – 3 балла;
    c) показать заинтересованным лицам нашу бета-версию – 1 балл;
    d) мы не выпускаем MVP – 0 баллов.

12. Мы демонстрируем время бесперебойной работы:
    a) на специальных общедоступных страницах – 5 баллов;
    b) в отчетах отдельным клиентам – 3 балла;
    c) в системе мониторинга – 1 балл;
    d) мы не демонстрируем – 0 баллов.

13. Мы устраняем инциденты в среде эксплуатации:
    a) путем быстрого пересоздания части инфраструктуры – 5 баллов;
    b) откатывая назад неудачные изменения – 3 балла;
    c) через процесс управления инцидентами – 1 балл;
    d) с помощью перезагрузки – 0 баллов.

14. Наш поток создания ценности:
    a) нарисован в состоянии «как есть» и спроектирован до состояния «как будет» – 5 баллов;
    b) нарисован в нашей памяти – 3 балла;
    c) нарисован на стене в кабинете руководителя – 1 балл;
    d) мы (ИТ-отдел) и есть ценность – 0 баллов.

15. Мы настраиваем нашу ИТ-инфраструктуру:
    a) исключительно скриптами, хранящимися в системе контроля версий – 5 баллов;
    b) скриптами, которые администраторы разработали для себя сами – 3 балла;
    c) вручную силами администраторов – 1 балл;
    d) вручную силами DevOps-инженеров – 0 баллов.

Нетрудно заметить, что всего в тесте 15 вопросов. Таким образом, максимально возможное количество баллов составляет 75. Будем считать, что:

- если вы набрали до 44 баллов – у вас отличные перспективы, со временем перерастающие в новые возможности интеграции с рынком труда;
- если вы набрали от 45 до 74 баллов – вы на верном пути, DevOps уже близко!
- если вы набрали 75 баллов – срочно свяжитесь с автором книги, у него к вам есть несколько уточняющих вопросов.

Список рекомендованной литературы

Среди сотен уже опубликованных к настоящему моменту книг по DevOps читать, как выясняется, особо и нечего. Нам предстоит многое узнать, многому научиться. Очень многие издания содержат весьма поверхностный взгляд на данную область. Другие фокусируются на отдельных аспектах явления, не охватывая всю ширину и не создавая целостной картины. Большое число очень интересных книг содержит информацию о смежных вопросах, затрагивая DevOps лишь косвенно. И к сожалению, многие книги отличаются довольно скромным соотношением объема полезной информации к числу строк.

Тем не менее для более глубокого изучения DevOps, помимо упоминавшихся в ходе изложения книг, рекомендуется обратить внимание на следующие публикации:

1) *Humble J., Farley D.* Continuous Delivery: Reliable Software Releases Through Build, Test, And Deployment Automation. 2011. ISBN 978-0-321-60191-9;
2) *Kim G., Humble J., Debois P., Willis J.* The Devops Handbook: How To Create Worldclass Agility Reliability And Security In Technology Organizations. 2016. ISBN 978-1-942-78800-3;
3) *Swartout P.* Continuous Delivery and DevOps – A Quickstart Guide. First published: November 2012. Second edition: December 2014. ISBN 978-1-784-39931-3;
4) *Gruver G.* Starting and Scaling DevOps in the Enterprise. 2016. ISBN 978-1-483-58358-7.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Олег Владимирович Скрынник

**DevOps для ИТ-менеджеров:
Концентрированное структурированное изложение
передовых идей**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 10,2375. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**