

**Джин Ким, Патрик Дебуа, Джон Уиллис, Джез Хамбл**

**Руководство по DevOps. Как добиться гибкости, надежности и безопасности мирового уровня в технологических компаниях**

## **Информация от издательства**

Научный редактор Николай Корытко

*Издано с разрешения IT Revolution Press LCC c/o Fletcher & Company и Andrew Nurnberg Associates International Ltd c/o ZAO "Andrew Nurnberg Literary Agency"*

*На русском языке публикуется впервые*

*Благодарим за помощь в подготовке издания Артема Каличкина, Дмитрия Зайцева, Михаила Чинкова, Виталия Рыбникова, Дениса Иванова, Валерия Пилия, Дмитрия Малыхина, Сергея Малютина, Александра Титова, Дениса Рыбака, Евгения Овчинцева, Алексея Климова, Игоря Авдеева*

## **Ким, Джин**

Руководство по DevOps. Как добиться гибкости, надежности и безопасности мирового уровня в технологических компаниях / Джин Ким, Патрик Дебуа, Джон Уиллис, Джез Хамбл; пер. с англ. И. Лейко и И. Васильева; [науч. ред. Н. Корытко]. — М.: Манн, Иванов и Фербер, 2018.

ISBN 978-5-00100-750-0

Профессиональное движение DevOps зародилось в 2009 году. Его цель — настроить тесные рабочие отношения между разработчиками программного обеспечения и отделами ИТ-эксплуатации. Внедрение практик DevOps в повседневную жизнь организации позволяет значительно ускорить выполнение запланированных работ, увеличить частоту релизов, одновременно повышая безопасность, надежность и устойчивость производственной среды. Эта книга представляет собой наиболее полное и исчерпывающее руководство по DevOps, написанное ведущими мировыми специалистами.

*Все права защищены.*

*Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.*

© 2016 by Gene Kim, Jez Humble, Patrick Debois, and John Willis

© Перевод, издание на русском языке, оформление. ООО «Манн, Иванов и Фербер», 2018

## **Предисловие кроссийскому изданию**

Впервые о DevOps заговорили в связи с переходом в эру цифровой экономики, когда скорость выпуска на рынок продуктов стала одним из ключевых конкурентных преимуществ. Технологиям, обеспечивающим стремительное развитие бизнеса, пришлось бежать со всех ног, чтобы только оставаться на месте, а для достижения дополнительных результатов, как минимум, в два раза быстрее. Компаниям понадобились инструменты для быстрого и непрерывного улучшения качества существующих процессов разработки продуктов и их максимальной автоматизации, потому что хороший продукт стал равен хорошему ИТ.

Свой путь погружения в DevOps я начала несколько лет назад, когда возглавила отдел тестирования системы подготовки регулярной банковской отчетности Neoflex Reporting, которая отличалась большим количеством параллельных веток разработки и обилием ручных процессов. В ее разработку к этому моменту уже были вложены десятки тысяч человеко-часов.

Засучив рукава, наша команда взялась за точечную автоматизацию этапов жизненного цикла продукта. В целом мы достигли неплохих результатов, но добиться слаженной и синхронной работы от всех участников процесса оказалось по-настоящему трудной задачей. Периодически возникающие «тут подкрутить», «там вручную запустить», «а это не на моей стороне», «я был на обеде», «исторически сложилось» тормозили ожидаемое от автоматизации ускорение.

Осознать, что же делать дальше, помогла книга, которую вы сейчас держите в руках. Мы прочитали её всей командой и здорово переработали текущие процессы взаимодействия в парадигме слаженности, простоты и удобства. А процессы сборки, развертывания инфраструктуры, установки, тестирования и выдачи поставки объединили в непрерывный производственный конвейер, вдохновленные идеей «все, что связано с кодом — тоже код». Довольно быстро были получены ошеломляющие результаты: время выпуска обновлений с одного дня сократилось до десятка минут, а работа над продуктом Neoflex Reporting стала приносить профессиональное удовольствие.

«Руководство по DevOps» — книга об эффективном ИТ настоящего. Захватывающий и понятный путеводитель, способный обобщить, разложить по нужным полочкам существующий опыт и обогатить его ценными идеями.

В книге описаны основные шаги и принципы построения производственного взаимодействия, автоматизации процессов и развития культуры разработки ПО. Теория щедро сдобрена историями реальных людей и компаний, прошедших непростой, но интересный путь к DevOps.

Неоспоримая ценность «Руководства...» в том, что оно помогает вырваться из рутины бытия и взглянуть на текущие процессы совершенно другими глазами. Приходит осознание того, что на точечных «костылях» автоматизации далеко не уйти, появляется понимание того, как выглядит путь роста и развития, который подходит именно вашей компании, проекту, продукту.

Желаю вам приятного чтения и пусть эта книга станет для вас источником неиссякаемого вдохновения!

*Лина Чуднова, руководитель практики DevOps компании «Неофлекс»*

## Введение

Путь к созданию книги «Руководство по DevOps» был долгим. Он начался в феврале 2011 г. с еженедельных переговоров по скайпу между соавторами. Мы решали, как создать руководство с рекомендациями — дополнение к книге *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*.

Прошло пять с лишним лет. Более двух тысяч часов работы. Книга «Руководство по DevOps» наконец завершена. В результате мы вознаграждены сполна, поскольку неожиданно обрели новое знание и поняли: сфера его применения гораздо шире, чем мы первоначально предполагали. Оно обеспечивает невиданные возможности. В конце концов мы сами воскликнули: «Ага!» — и нам кажется, что многие читатели разделят наше мнение.

Мне повезло: с 1999 г. я изучал организации, использующие высокопроизводительные технологии. Вот один из моих первых выводов: решающее значение для успеха имеет перекрестное взаимодействие функциональных групп, занимающихся эксплуатацией, информационной безопасностью и разработкой. Я до сих пор помню, как впервые осознал масштабы нисходящей спирали, в которую заключена деятельность этих групп с их противоположными задачами.

Это было в 2006 г., и мне тогда представилась возможность поработать целую неделю с группой, решавшей отданые на аутсорсинг ИТ-задачи, поставленные крупной службой резервирования и продаж авиабилетов. Участники группы рассказали об увеличивающихся негативных последствиях ежегодных крупных обновлений программного обеспечения: каждый раз наступал настоящий хаос, шквал неудобств как для исполнителей, так и для заказчика. Из-за простоеов у пользователей им приходилось выплачивать немалые компенсации согласно договорам по сервисному обслуживанию. Увольнялись наиболее способные и опытные работники, так как, опасаясь потерять прибыль, компания вынуждала их наращивать темп, выполнять массу незапланированной работы и «тушить пожары». У оставшегося персонала не хватало сил справляться со все возрастающим потоком требований заказчиков, желавших исправления ошибок. От расторжения сервисного контракта компанию спасали только героические усилия менеджеров среднего звена, и все были уверены: у контракта нет будущего, его не продлят на следующие три года.

Отчаяние и безнадежность подтолкнули меня к тому, чтобы начать нечто вроде наступательной операции. Разработка всегда рассматривалась как часть стратегии, а эксплуатация — тактики. Нередко они частично или даже полностью отдавались на аутсорсинг, чтобы лет через пять вернуться обратно, еще более усложнившимися.

Многие годы мы размышляли, как улучшить ситуацию. Вспоминаю, как на конференции Velocity Conference 2009 с интересом следил за обсуждением фантастических результатов, достигнутых благодаря использованию принципов бизнес-архитектуры, технических методов и норм корпоративной культуры в совокупности. Теперь эта методика известна нам как DevOps. Тогда я неподдельно взодновался: передо мной наметился путь выхода из создавшейся ситуации — его-то мы так долго искали. Стремясь распространить новое знание как можно шире, я и решил выступить соавтором *The Phoenix Project*. Вы запросто можете представить себе, какое огромное внутреннее удовлетворение я испытал, видя видя отзывы людей о том, как книга помогла им прийти к озарению и воскликнуть: «Ага!»

Мое личное «Ага!» впервые раздалось в 2000 г., в стартапе, который был моим первым местом работы после окончания обучения. Некоторое время нас, технических специалистов, было только двое. Поэтому мне приходилось заниматься всем: сетями, программированием, поддержкой пользователей, системным администрированием. Мы выпускали ПО, размещая его на FTP прямо с рабочих станций.

В 2004 г. я перешел в консалтинговую компанию ThoughtWorks, где впервые принял участие в работе над проектом в составе команды численностью около 70 человек. Я входил в группу из восьми инженеров, занимавшуюся развертыванием нашей программы в среде, приближенной к производственной. Поначалу задание вызывало у нас сильный стресс. Но спустя несколько месяцев мы перешли от режима работы вручную, занимавшего около двух недель, к автоматическому разворачиванию продолжительностью всего один час. Теперь можно было за доли секунды откатывать конфигурации назад и вперед, используя технику «Blue-Green разворачивания» в рабочее время.

Проект породил массу идей, изложенных как в этой книге, так и в другой — «Непрерывное развертывание ПО». Они убедили меня и многих моих коллег: с какими трудностями нам ни пришлось бы сталкиваться, мы способны действовать с максимальной отдачей и помогать людям.

Для меня это была целая цепь событий. В 2007 г. я работал в проекте по миграции data-центра совместно с несколькими Agile-командами. Я завидовал их высокой продуктивности, умению выполнять большой объем работы за ограниченное время.

Получив следующее задание, я приступил к эксперименту по внедрению методики канбан в работу группы эксплуатации и увидел: группа стала быстро меняться. Позже, на конференции Agile Toronto 2008, я представил свой доклад в IEEE на эту тему, но, к сожалению, он не получил широкого отклика в Agile-сообществе. Мы начали создавать административную группу для системы Agile, но тут я переоценил значение человеческого фактора.

Увидев на Velocity Conference 2009 презентацию Джона Олспоу и Пола Хаммонда «10 развертываний в день», я убедился: у меня есть единомышленники. Поэтому я решил организовать первую конференцию DevOpsDays и так случайно создал новый термин — DevOps.

Энергетика мероприятия была уникальной. Участники благодарили меня, утверждая, что конференция изменила их жизнь к лучшему. Так я осознал степень воздействия концепции DevOps и с тех пор неустанно продвигаю ее.

В 2008 г. я продал свою консалтинговую компанию, специализировавшуюся на внедрении крупномасштабных устаревших решений в области управления конфигурациями и мониторинга (Tivoli). Тогда же я впервые встретил Люка Канисса (основателя компании PuppetLabs). Люк выступал с презентацией о Puppet на проводившейся издательством O'Reilly конференции по конфигурационному управлению (СМ) на основе открытого исходного кода.

Поначалу я скучал на заднем ряду лекционного зала, размышляя, что нового этот двадцатилетний парень может рассказать мне об управлении конфигурациями. Ведь я занимался этим всю жизнь, помогая крупнейшим корпорациям мира разрабатывать решения в области СМ и других сферах управления эксплуатацией. Однако через пять минут после начала доклада я уже сидел на первом ряду. Я тут же понял: все, что я делал за последние 20 лет, я делал неправильно. Люк описывал то, что я сейчас называю вторым поколением СМ.

После доклада мне удалось поболтать с ним за чашечкой кофе. Я был совершенно восхищен идеей, сейчас называемой «инфраструктура как код». Люк увлекся и стал подробно объяснять, что имеет в виду. Он верит, что эксплуатация становится похожей на разработку программ. Специалисты отдела эксплуатации хотят, чтобы конфигурации проверялись системой контроля качества и чтобы в рабочий процесс были адаптированы методики обеспечения CI/CD. Поскольку я к тому времени уже немало проработал в области эксплуатации ИТ, я ответил ему примерно так: «Это то же, что пытаться заставлять управленцев петь как Led Zeppelin».

Я жестоко ошибался.

Примерно через год на другой конференции Velocity, проводившейся в 2009 г. O'Reilly, я увидел презентацию Эндрю Шефера по инфраструктуре Agile. В ней он показал ставший каноническим рисунок — метафорическую стену между разработчиками и инженерами эксплуатации, через которую они перебрасывают друг другу рабочие задания. Он назвал это «стеной неразбериши». Идеи, высказанные в презентации, в систематизированном виде выражали то, что Люк пытался рассказать мне годом ранее. Для меня это стало откровением. В том же году меня, единственного из американцев, пригласили на первую конференцию DevOpsDays в Генте. Ко времени окончания конференции идея, ныне получившая название DevOps, полностью овладела моим разумом.

Из всего вышесказанного ясно, что соавторы книги пришли к единому выводу, хотя шли к нему разными путями. Реальность доказала, что описанная проблема существует практически везде и решения с помощью DevOps применимы повсюду.

Цель написания этой книги — показать, как воспроизвести DevOps-трансформации, частью которых мы были или которые мы наблюдали со стороны. Еще мы хотим развеять множество мифов о том, почему DevOps не будет работать в тех или иных ситуациях. Нам довелось слышать примерно следующее.

**Миф 1:**DevOps пригоден только для стартапов. Методы DevOps впервые были применены единорогами интернет-индустрии: Google, Amazon, Netflix и Etsy. Каждая из компаний в определенные моменты своей истории рисковала выпасть из бизнеса из-за проблем, обычно возникающих в традиционных организациях (их еще называют рабочими лошадками экономики). Это опасные релизы, приводящие компанию к катастрофическому провалу, неумение быстро проводить изменения продукта или сервиса, чтобы превзойти конкурентов в новой области, проблемы с соблюдением нормативных требований, неспособность масштабироваться, высокая степень недоверия между разработкой и эксплуатацией и так далее.

Однако каждая из названных организаций смогла преобразовать свою архитектуру, технические методы, производственную культуру и достичь выдающихся результатов благодаря DevOps. Как язвительно заметил известный американский специалист по информационной безопасности Бранден Вильямс, «пусть больше не будет разговоров о пегасах или рабочих лошадках в DevOps,

пусть останутся только чистокровные скакуны, а остальные отправятся на мыловарню в качестве сырья».

**Миф 2:** *DevOps заменяет собой Agile*. Принципы и методы DevOps совмещаются с Agile, причем многие отмечают, что DevOps — логическое продолжение Agile. Agile часто оказывается эффективным катализатором DevOps, поскольку предпочитает организовывать деятельность небольших команд, непрерывно поставляющих пользователям код высокого качества.

Многие практики DevOps возникают, если мы продолжаем управлять нашей работой за пределами цели «код, потенциально пригодный для релиза» в конце каждой итерации, расширяя ее до того, чтобы наш код всегда находился в развертываемом состоянии, а разработчики ежедневно синхронизировали свои изменения с основной веткой кода и могли продемонстрировать новые изменения в окружениях, близким к реальным.

**Миф 3:** *DevOps несовместим с ITIL*. Многие рассматривают DevOps как ответ на ITIL или ITSM (управление ИТ-инфраструктурой компании). Его описание было впервые опубликовано в 1989 г. ITIL сильно повлияла на несколько поколений практиков в области управления инфраструктурой, включая одного из авторов этой книги. Это постоянно развивающаяся библиотека методов, позволяющих кодифицировать процессы и практики, лежащие в основе признанных во всем мире способов управления ИТ, связывающих воедино стратегию услуг, разработку и поддержку.

Методы DevOps можно сделать совместимыми с процессами ITIL. Однако для обеспечения более короткого цикла разработки и повышения частоты развертываний, предложенных DevOps, многие области процессов ITIL должны быть полностью автоматизированы. Следует решить проблемы, связанные с процессами управления конфигурациями и релизами (например, как обеспечивать постоянную готовность базы управления и библиотеки программ). DevOps требует, чтобы возникающие ошибки быстро обнаруживались и устранились, поэтому дисциплина применения ITIL в проектировании архитектуры, обработке сбоев, решении проблем остается актуальной, как и всегда.

**Миф 4:** *DevOps несовместим с требованиями информационной безопасности*. Отсутствие традиционных методов контроля (например, разделение ответственности, изменение процессов проверки кода, проверка безопасности вручную по окончании проекта) может вызвать тревогу у специалистов по безопасности.

Однако это не означает, что в организациях, использующих DevOps, отсутствует эффективный контроль. Вместо работ по обеспечению безопасности и проверки соответствия требования ИБ, проводящихся на завершающем этапе проекта, необходимые проверки интегрированы в каждую стадию ежедневного цикла на протяжении всего цикла разработки. В результате обеспечиваются более высокое качество и безопасность.

**Миф 5:** *DevOps означает отсутствие необходимости управления ИТ-эксплуатацией, то есть NoOps (дословно — «нет эксплуатации»)*. Многие неправильно трактуют DevOps как полное исключение необходимости ИТ-эксплуатации. Однако такое утверждение редко бывает справедливо. Хотя характер оперирования может измениться, само управление остается важным как никогда. Просто оно на гораздо более ранних этапах жизненного цикла ПО взаимодействует с разработкой, продолжающей действовать параллельно с ИТ-эксплуатацией еще долго после того, как разработанный код развернут в производственной среде.

Вместо того чтобы отдел эксплуатации разгребал поступающие заявки вручную, DevOps дает разработчикам возможность делать большинство операций через API и самообслуживающиеся платформы, такие как: создание среды, тестирование и развертывание кода, получение и отображение метрик о ПО и т. д. Когда реализован такой подход, ИТ-эксплуатация становится похожей на процесс разработки (то же справедливо для управления качеством и обеспечения информационной безопасности), сцепленный с разработкой продукта, где под продуктом понимается платформа, используемая, чтобы надежно, быстро и безопасно тестировать ИТ-сервисы, развертывать их и запускать в производственной среде.

**Миф 6:** *DevOps — это просто реализация подхода «инфраструктура как код»*. Хотя многие из практик и подходов DevOps, приведенных в этой книге, требуют автоматизации, для реализации DevOps также необходимо изменение архитектуры системы и культуры производства, дающее

возможность достичь общих целей в ходе работы по повышению создаваемой ценности ИТ. Это выходит далеко за рамки простой автоматизации. Как написал Кристофер Литл, один из самых первых летописцев DevOps, «это не автоматизация, так же как астрономия — это не телескопы».

**Миф 7:** *DevOps применим только к программам с открытым исходным кодом.* Хотя многие случаи успешного внедрения DevOps действительно имели место в организациях, использовавших ПО, входившее в группу LAMP (Linux, Apache, MySQL, PHP), имелись истории успеха, и не зависевшие от использованных технологий. Например, в приложениях, написанных на , коболе, языке ассемблера мейнфреймов, а также в системах SAP и даже в коде встроенных систем (например, микропрограммное обеспечение принтеров HP LaserJet).

Каждый из авторов воодушевился удивительными инновациями, реализованными сообществом DevOps, и достигнутыми результатами: были созданы надежные системы, позволившие небольшим командам быстро и независимо разрабатывать и проверять код, который может быть без риска развернут у заказчика. Учитывая нашу убежденность, что DevOps — воплощение методов создания динамичных, обучающихся организаций, постоянно укрепляющих культуру высокого доверия, представляется неизбежным, что эти организации будут продолжать инновации и выйдут победителями на рынке.

Мы искренне надеемся, что книга «Руководство по DevOps» станет ценным источником. Как руководство по проведению DevOps-трансформации. Как набор практических примеров для накопления опыта. Как летопись истории DevOps. Как средство для организации коалиции и достижения общих целей владельцев продукта, архитекторов, разработчиков, инженеров контроля качества, эксплуатации и информационной безопасности. Она подскажет, как получить максимальную поддержку со стороны руководства при внедрении инициатив DevOps, как сформировать нравственный императив для изменения способов управления технологическими организациями при обеспечении высокой эффективности. Она поможет создать более оживленную и дружелюбную рабочую среду, чтобы любой участник смог учиться в течение всей жизни — это не только поможет каждому исполнителю достичь целей, но и приведет организацию к победе.

## **Предисловие**

В прошлом многие сферы инженерной деятельности пережили серьезные изменения, за счет чего она сама стала понятнее. И хотя существуют университетские курсы и компании, осуществляющие техническую поддержку в каждой конкретной области (гражданском строительстве, механике, электроснабжении, атомной технике и т. д.), современное общество нуждается во всех перечисленных направлениях, чтобы по достоинству оценить их полезность и развивать междисциплинарные направления.

Вот, например, конструирование современного автомобиля. Где кончается область компетентности инженера-механика и начинается зона ответственности инженера-электрика? Где (и как, и когда) специалист по аэродинамике (безусловно, имеющий четкое представление о форме, размере и местах размещения окон) должен взаимодействовать с экспертом в области эргономики пассажиров? Что можно сказать о химическом влиянии топливной смеси и масел на материалы, из которых изготовлены двигатель и трансмиссия, в течение всего времени эксплуатации машины? Во время конструирования автомобиля можно задать много других вопросов, но конечный результат одинаков: для успеха современных технических проектов абсолютно необходимы две вещи — умение рассмотреть проблему с разных точек зрения и опыт совместной деятельности.

Чтобы направление деятельности или область знаний развивались и крепли, необходимо достичь того уровня, когда появляется возможность серьезно задуматься, как зародилось это направление или область, отыскать перспективы развития и объединить все это, желая понять облик общества будущего.

В этой книге представлен способ такого синтеза. Ее следует рассматривать как стартовый набор перспектив в области разработки и эксплуатации ПО (я по-прежнему считаю эту сферу деятельности растущей и быстро развивающейся).

Независимо от того, в какой отрасли вы работаете, какой продукт выпускаете, какие услуги оказывает ваша организация, данный образ мышления имеет первостепенное значение: он просто необходим для выживания любой компании-лидера в любой области бизнеса и технологий.

*Джон Олспоу, директор по технологиям компании Etsy*

*Бруклин, Нью-Йорк, август 2016 г.*

## **Вступление. Как будет выглядеть мир, если разработка и эксплуатация пойдут по принципу DevOps**

Представьте себе мир, в котором владельцы продукта, разработчики, тестировщики, сотрудники ИТ-эксплуатации и специалисты по информационной безопасности действуют сообща, не только помогая друг другу, но и обеспечивая будущий успех организации. Трудясь ради общей цели, они обеспечивают быстрое внедрение плановых результатов в производство (выполняя в день десятки, сотни или даже тысячи развертываний кода) и достигают при этом высокого уровня стабильности, устойчивости, доступности и безопасности.

В таком мире кросс-функциональные группы, несомненно, выполняют проверку своих предположений, какие именно функции особенно порадуют пользователей и послужат достижению целей организации. Они не просто заботятся о реализации функций, нужных пользователям, но также активно обеспечивают бесперебойную работу и проверку цепочки создания ценностей, не вызывая при этом хаоса в управлении ИТ-эксплуатацией и сбоев у любых внутренних и внешних клиентов.

В то же самое время тестировщики, сотрудники эксплуатации и специалисты по информационной безопасности постоянно стараются уменьшить взаимные трения внутри команды разработчиков, создавая системы, дающие возможность действовать продуктивнее и результативнее. Когда компетентность специалистов по качеству, сотрудников ИТ-эксплуатации и специалистов по информационной безопасности становится доступной командам, занимающимся поставками, автоматизированными инструментами и платформами самообслуживания, они могут использовать все это в повседневной работе и перестать зависеть от других команд.

Такой подход дает организациям возможность создать надежную систему: небольшие команды быстро и автономно разрабатывают, тестируют и развертывают код и вдобавок делают это надежно и безопасно. Это позволяет организациям максимизировать продуктивность разработчиков, организовать обучение внутри организаций, обеспечить высокую удовлетворенность исполнителей и стать победителем в конкурентной рыночной борьбе.

Таковы результаты использования DevOps. Но большинство из нас живет совсем в другом мире. Зачастую системы, в которых мы работаем, несовершенны, демонстрируют очень низкие результаты и не позволяют раскрыть наш истинный потенциал. В нашем мире отделы разработки и ИТ-эксплуатации враждуют; тестирование и обеспечение информационной безопасности проводятся только ближе к окончанию проекта, то есть слишком поздно для устранения проблем. И почти любая серьезная деятельность требует от сотрудника значительных усилий, выполнения вручную ряда последовательных задач, из-за чего занятые поиском и устранением проблем инженеры и конечные пользователи просто ждут, пока кто-то сделает свою часть. Это не просто замедляет получение результата, но и вносит хаос, особенно в процесс развертывания, что приводит к негативным последствиям как для клиентов, так и для бизнеса.

В результате мы оказываемся далеко от желаемых целей. Все в организации недовольны уровнем производительности ИТ-подразделений. Итог — бюджет урезан, все разочарованы, но понимают, что бессильны изменить ход процесса разработки и добиться заметных результатов. Каким же должно быть решение? Необходимо изменить методы работы, и DevOps — это наилучший способ продвижения вперед.

Чтобы лучше понять потенциал DevOps, давайте рассмотрим промышленную революцию 1980-х гг. Внедрив принципы и методы бережливого производства (Lean), промышленные компании значительно улучшили производительность предприятий, сократили время выполнения заказов, повысили удовлетворенность своих потребителей, что позволило им стать победителями на рынке.

Раньше среднее время выполнения заказа заводом-изготовителем составляло шесть недель, причем без нарушения сроков выполнялось менее 70 % заказов. К 2005 г. благодаря широкому внедрению методов бережливого производства временной показатель сократился менее чем до трех недель, и 95 % заказов выполнялись точно в срок. Организации, не внедрившие принципы бережливого производства, потеряли долю на рынке, а многие вообще ушли из бизнеса.

Точно так же повысились требования к продуктам: то, что было хорошо в предшествующие десятилетия, перестало удовлетворять заказчиков. Следующие 40 лет стоимость и время разработки и внедрения стратегических возможностей для бизнеса снижались все активнее. В 1970-1980-х гг. разработка и внедрение большинства новых технологий требовали от одного до пяти лет и часто обходились в десятки миллионов долларов.

К 2000-м гг. благодаря развитию технологий и внедрению принципов и методов Agile время, необходимое для разработки новой функциональности, уменьшилось до месяцев или даже недель. Но и сам процесс внедрения также занимал недели или месяцы, причем часто с неприемлемыми результатами.

Но к 2010 г. в связи с внедрением DevOps и непрекращающейся коммодитизацией компьютерного оборудования, программ, а теперь и облачных технологий новые функции (и даже целые компании-стартапы) могут создаваться за недели и быстро — за часы или даже за минуты — внедряться в производство. Для таких организаций развертывание стало рутинной операцией, практически не содержащей рисков. Появилась возможность проводить эксперименты для проверки бизнес-идей, выясняя, какие из них наиболее ценные для клиентов и самой организации и какие можно быстро превратить в «фичи», а уже их, в свою очередь, быстро и без рисков развернуть на производстве.

Таблица 1. Тенденция к более быстрой, дешевой и имеющей мало рисков поставке программного обеспечения

	1970–1980 гг.	1990-е гг.	С 2000-х гг. до наших дней
Эпоха	Майнфреймы	Клиент-сервер	Коммодитизация и облака
Типичные технологии эпохи	COBOL, DB2 в MVS и пр.	C++, Oracle, Solaris и т.д.	Java, MySQL, Red Hat, Ruby on Rails, PHP и т.д.
Продолжительность цикла разработки и внедрения	1–5 лет	3–12 месяцев	2–12 недель
Стоимость, млн долл. США	1–100	0,1–10	0,01–1
Зона риска	Вся компания	Линейка продуктов или отдел компании	Функциональность продукта
Цена ошибки	Банкротство, продажа компании, массовые сокращения	Потеря прибыли, увольнение руководства	Пренебрежимо мала

Источник: Презентация «Быстрота и объем (Скорость выигрывает)» Адриана Кокрофта на конференции FlowCon в Сан-Франциско, ноябрь 2013 г.

Организации, внедрившие принципы и методы DevOps, сейчас нередко за день выполняют сотни, а то и тысячи развертываний. В эпоху, когда для получения конкурентных преимуществ требуется быстрый выход на рынок и непрекращающееся экспериментирование, компаниям, неспособным показать такие же результаты, суждено уступить свою долю рынка более гибким и легким на подъем конкурентам и даже полностью уйти из бизнеса, подобно промышленным предприятиям, своевременно не внедрившим принципы бережливого производства.

Сегодня независимо от того, к какой отрасли относится компания, приобретение клиентов и предоставление им создаваемой ценности зависит от технологичности канала поставки этой ценности. Джон Иммелт, исполнительный директор компании General Electric, выразил эту мысль сжато и точно: «Каждая отрасль промышленности и каждая компания, не желающие делать программное обеспечение центром бизнес-стратегии, не имеют будущего». Или, как сказал Джон Сновер, технический специалист корпорации Microsoft, «в предыдущие экономические эпохи коммерческие предприятия создавали ценности, перемещая атомы. Теперь они делают это, перемещая биты».

Трудно переоценить масштабы этой проблемы. Она есть в каждой организации независимо от отрасли, размера компании и ее профиля — коммерческого или нет. Сейчас чаще чем когда-либо то, как идет управление и выполняется технологическая работа, определяет главное: победит ли организация на рынке и, более того, выживет ли она. Во многих случаях нам придется освоить принципы и методы, на первый взгляд разительно отличающиеся от тех, которыми мы успешно руководствовались в предыдущие десятилетия (см. ).

Итак, теперь, когда мы обосновали неотложность проблемы, давайте уделим некоторое время более подробному изучению ее симптомов и тому, почему она возникает и почему с течением времени только обостряется, если не принять серьезные меры.

Большинство компаний не в состоянии развернуть производственные изменения в реальной среде в течение нескольких минут или часов, им для этого потребуются недели или даже месяцы. Они также не могут доставлять в рабочую среду сотни или тысячи изменений в день, вместо этого они изо всех сил пытаются произвести развертывание ежемесячно или хотя бы ежеквартально. Также не предусматривается развертывание производственных процессов, вместо этого — перебои в работе и хронически героические усилия, направленные на ликвидацию проблем.

В эпоху, когда для получения конкурентного преимущества надо быстро выпускать продукты на рынок, обеспечивать высокий уровень поддержки и неустанно экспериментировать, такие организации оказываются в невыгодном для конкуренции положении. В значительной степени это обусловлено неспособностью разрешить коренной, хронический конфликт в их технологической организации.

Почти в любой ИТ-компании существует постоянный конфликт между разработкой и ИТ-эксплуатацией, что создает нисходящую спираль и приводит к постоянному увеличению времени, необходимого для выпуска на рынок новых продуктов или новых функциональностей, снижению качества и, что самое плохое, к постоянному увеличению технического долга.

Термин «технический долг» был впервые предложен Уордом Каннингемом. Подобно финансовому, технический долг — решения, необходимые для ликвидации проблем, с течением времени становящихся все более трудно разрешимыми при постоянном уменьшении будущих возможностей для маневра. Даже действуя благородно, мы все равно вынуждены выплачивать проценты.

Один из факторов, вызывающих такое состояние, — часто проявляющаяся конкуренция между разработчиками и ИТ-эксплуатацией. Организации, специализирующиеся в области информационных технологий, должны отвечать за многое. Среди прочих есть две цели, и они должны достигаться одновременно:

- реагировать на быстро меняющийся конкурентный ландшафт;
- обеспечить стабильный, надежный и безопасный сервис для клиента.

Нередко разработчики берут на себя ответственность за реагирование на изменения на рынках, развертывание новой функциональности и изменений в реальной среде в кратчайшие сроки. Отдел ИТ-эксплуатации готов отвечать за предоставление заказчикам стабильных, надежных и безопасных ИТ-услуг, делая при этом затруднительным или даже невозможным внесение каких-либо изменений, ставящих производство под угрозу. При такой ситуации разработчики и отдел ИТ-эксплуатации преследуют абсолютно противоположные цели и имеют разные стимулы.

Доктор Элияху Голдратт, один из создателей методологии управления производством («теории ограничений»), называл конфигурации такого типа «корневым, хроническим конфликтом». Он заключается в том, что корпоративное нормирование и стимулы в разных подразделениях препятствовали достижению глобальных целей организации.

Этот конфликт настолько сильно препятствует результативности в бизнесе, как внутри ИТ-организаций, так и вне их, что возникает нисходящая спираль. Хронические конфликты зачастую ставят технических специалистов в условия, приводящие к созданию негодного ПО, низкому качеству поддержки, плохим результатам у заказчиков. А еще к тому, что практически ежедневно приходится искать обходные пути для решения проблем и незамедлительно прилагать героические усилия по внесению исправлений в отделах управления производством, разработки, тестирования, ИТ-эксплуатации или информационной безопасности (см. ).

У этой драмы три акта, вероятно, знакомые всем, кто имеет отношение к сфере ИТ.

Первый акт начинается в отделе ИТ-эксплуатации, где главная цель — сохранение работоспособности приложений и инфраструктуры, чтобы организация могла передавать клиентам

продукцию. В повседневной деятельности многие проблемы вызваны тем, что приложения и инфраструктура оказываются сложными, плохо документированными и невероятно хрупкими. Это технический долг и ежедневный поиск обходных решений. С ними приходится постоянно сталкиваться, выслушивая клятвы, что кавардак будет обязательно ликвидирован, как только появится немного свободного времени. Но таковое, естественно, никогда не появляется.

Вызывает тревогу то обстоятельство, что наиболее хрупкие творения обеспечивают поддержку наиболее важных систем, создающих доход, или особенно серьезных проектов. Иными словами, именно системы, чаще всего выходящие из строя, наиболее важны для нас, и, кроме того, именно на них сильнее всего оказываются внесенные нами срочные изменения. И когда эти корректировки приводят к сбою, они ставят под удар наиболее важные обязательства организации, такие как доступность для клиентов, цели по получению дохода, безопасность данных пользователей, точность финансовых отчетов и так далее.

Второй акт начинается, когда кто-то должен компенсировать невыполненные обязательства — это может быть менеджер продукта, обещающий реализовать более впечатляющие возможности, чтобы восхитить заказчиков, или директор, предлагающий более агрессивные цели по доходности. Затем, забыв о реальных возможностях технологии или о том, из-за каких обстоятельств не были выполнены предыдущие обязательства, руководство заставляет технические отделы любой ценой реализовать эти новые обещания.

В результате перед разработчиками ставится задача срочно создать проект, что неизбежно требует решения новых технических проблем и «удаления всего лишнего», чтобы успеть реализовать задачу в срок. Так увеличиваются обязательства по техническому долгу, а параллельно звучат привычные обещания исправить все проблемы, как только появится немного свободного времени.

Такое решение подготавливает сцену для третьего и последнего акта пьесы, когда все необходимые действия становятся более и более трудными: каждый исполнитель все глубже увязает в выполнении задания, коммуникации между сотрудниками и отделами становятся медленными, а очередь из заданий на выполнение понемногу растет. Работа затягивается узлом, небольшие действия вызывают большие сбои, все начинают проявлять опасение и нетерпимость к изменениям. Нужно больше обсуждений, координации и одобрения различными инстанциями. Группы сотрудников чаще вынуждены ждать, когда будет выполнена задача, задерживающая их собственные действия, а качество при этом только ухудшается. Дело движется все медленнее, и, чтобы увеличить скорость, приходится прилагать больше усилий (см. ).

В настоящем разглядеть нисходящую спираль сложно, но ретроспективно она видна отчетливо. Мы начинаем замечать, что развертывание готового кода занимает все больше времени: вместо минут — часы, дни и даже недели. Но хуже всего то, что результаты развертывания оставляют желать лучшего, а это ведет к возрастанию времени простоев у клиентов, что, в свою очередь, требует от отдела ИТ-эксплуатации героических усилий по «тушению пожаров», отнимающих у него возможность выплатить технический долг.

В результате циклы поставки продукта затягиваются все сильнее, новых начинаний все меньше, а проекты, которые все-таки появляются, оказываются менее амбициозными. Кроме того, обратная связь, доносящая результаты от каждого — особенно поступающая от клиентов, — ослабевает и становится более медленной. Несмотря на все усилия, ситуация только ухудшается: мы уже не в состоянии быстро реагировать на изменяющуюся ситуацию на рынке, предоставляя стабильный и надежный сервис клиентам. В результате мы теряем свою долю рынка.

Снова и снова повторяется одно и то же: если терпит неудачу подразделение ИТ, то провал ждет всю организацию. Как пишет в своей книге *The High-Velocity Edge* Стивен Спир, неважно, наступают ли печальные последствия «медленно, как постепенно развивающаяся болезнь», или происходят быстро, «как пожар дома... разрушение в обоих случаях полное».

Более десяти лет авторы этой книги наблюдали нисходящую спираль в огромном количестве организаций всех типов и размеров. И в конце концов поняли, из-за чего она появляется и почему для ее устранения необходимо использование принципов DevOps. Во-первых, как уже говорилось, каждая ИТ-компания имеет две противоположные цели, а во-вторых, любая такая организация — технологическая, понимает она это или нет.

По словам Кристофера Литла, одного из первых летописцев DevOps, «каждая компания — технологическая, независимо от того, к какой области бизнеса она себя причисляет. Банк — это просто ИТ-организация с банковской лицензией».

Чтобы убедиться в верности этих слов, примите во внимание, что большинство крупных проектов связаны со сферой ИТ. Как говорится, почти невозможно принять какое-либо бизнес-решение, не вызывающее хотя бы одного изменения в работе ИТ-подразделений.

В деловом и финансовом мире проекты важны, поскольку служат основными двигателями

изменений внутри организаций. Проекты должны получить одобрение руководства, для них утверждается бюджет, за расходы нужно отчитываться, поэтому проекты — способ осуществить любые цели и ожидания организации, то есть и роста, и сокращения.

Проекты обычно финансируются за счет вложения капитала (например, заводы: закупка оборудования, главные части проектов и расходы капитализируются и окупаются спустя годы). 50 % в наше время тратится на технические нужды. Это справедливо даже для вертикальных линеек так называемой низкотехнологичной промышленности, то есть тех отраслей, где исторически сложилось так, что вклады в разработку технологий невысоки (энергетика, металлургия, ресурсодобывающие отрасли, автомобилестроение и строительство). Иными словами, это отрасли, в которых руководителям требовалось опираться на эффективное управление IT-отделами ровно настолько, насколько это нужно для достижения их целей.

Когда сотрудники, особенно занимающие невысокую должность в отделе разработки, несколько лет находятся в ловушке нисходящей спирали, они зачастую чувствуют себя заложниками системы, запрограммированной на неудачи, и понимают, что не в силах достичь заметных результатов. Ощущение бессилия нередко сменяется эмоциональным выгоранием, ощущением покорности судьбе, цинизмом и даже безнадежностью и отчаянием.

Психологи утверждают: создание системы, порождающей чувство бессилия, — одна из основных опасностей, которым мы подвергаем своих близких. Мы лишаем других возможности управлять собственными достижениями, создавая культурную среду, где подчиненные опасаются поступать правильно из-за страха наказания, неудачи или риска потерять средства к существованию. В результате формируются условия для появления *приобретенной беспомощности*: инженеры теряют желание или способность поступать так, чтобы избежать подобных проблем в будущем.

Для сотрудников это означает сверхурочные, работу по выходным и ухудшение качества жизни, причем не только их самих, но и всех тех, кто от них зависит, в том числе членов семьи и друзей. Неудивительно, что, когда складывается подобная ситуация, мы теряем лучших (исключая тех, кто обладает гипертрофированным чувством ответственности или связан какими-то обязательствами).

Помимо описанных неудобств, существующие способы действия провоцируют также финансовые потери, а ведь их можно было избежать. Можно оценить такие издержки в 2,6 триллиона долларов в год. На момент написания книги — сумма, равная валовому внутреннему продукту Франции (как считается, шестой экономики в мире).

Предлагаем следующие расчеты: по оценкам компании IDC и компании Gartner, примерно 5 % общемировой суммы валового внутреннего продукта (3,1 триллиона долларов) тратятся на IT-отрасли (оборудование, услуги, телекоммуникации). Если считать, что 50 % этой суммы ушли на операционные расходы и поддержание существующих систем, а треть этих 50 % была потрачена на незапланированные работы и переделку, то впustую было потрачено примерно 520 миллиардов долларов.

Если применение методов DevOps поможет уменьшить потери за счет лучшего управления и повышения качества результата и увеличить потенциал сотрудников в пять раз (и это по самым скромным подсчетам), то можно получить дополнительно 2,6 триллиона долларов в год.

В предыдущих разделах мы описали проблемы и отрицательные последствия существующего положения вещей, вызванного хроническим конфликтом, — начиная от неспособности организации достичь целей и заканчивая вредом, причиняемым нам. DevOps решает эти проблемы, одновременно позволяя увеличить производительность организации в целом, обеспечить достижение различными отделами (например, разработчиками, контролем качества, IT-эксплуатацией, отделом информационной безопасности) своих функциональных целей и улучшить условия для сотрудников.

Такое удивительное сочетание наглядно объясняет, почему DevOps вызывает восхищение и энтузиазм и так быстро завоевывает сторонников, включая лидеров в области технологий, инженеров и других участников значительной части экосистемы разработки ПО.

В идеале небольшие команды разработчиков трудятся независимо, разрабатывая функциональности, проверяя их правильность в среде, приближенной к реальной, и быстро, надежно и безопасно развертывая их в производственной среде. Развертывание кода — рутинная и предсказуемая процедура. Вместо того чтобы начинать развертывание поздно вечером в пятницу и тратить все выходные, его можно выполнять в разгар рабочего дня, когда все сотрудники на своих местах. При этом они даже не будут замечать развертывания — за исключением случаев, когда обнаружат появление новых функций или исчезновение ошибок, что приведет их в восторг. К тому же, осуществляя развертывание кода в середине рабочего дня, сотрудники IT-эксплуатации в первый раз за долгие десятилетия получат возможность трудиться как все нормальные люди — в рабочее время!

Благодаря созданию контуров быстрой обратной связи на каждом этапе процесса любой исполнитель имеет возможность сразу же увидеть результат своих действий. Когда изменения зафиксированы в системе контроля версий, быстрые автоматизированные тесты проводятся в среде, близкой к производственной, снова и снова подкрепляя уверенность, что и код, и среда работают как запланировано, всегда безопасны и готовы к развертыванию.

Автоматизированное тестирование дает разработчикам возможность быстро обнаруживать ошибки (обычно за минуту), что позволяет сразу же их исправлять и учиться тому, что невозможно работать нормально, если ошибка обнаруживается спустя шесть месяцев после интеграционного тестирования, когда связи между причиной и эффектом уже давно выветрились из головы. Вместо того чтобы накапливать технический долг, следует устранять проблемы сразу после обнаружения, если необходимо — с привлечением всей организации, поскольку ее глобальные цели перевешивают локальные цели группы или даже отдела.

Всеобъемлющий сбор телеметрии о коде и программной среде обеспечивает своевременное обнаружение проблем и их быстрое исправление. Он подтверждает: все на месте, как предусмотрено, и клиенты получают продукт благодаря предоставленному нами ПО.

При таком сценарии каждый чувствует себя на своем месте: архитектура процесса дает возможность небольшим командам действовать уверенно, даже не будучи тесно привязанными к работе, выполняемой другими командами, и использовать платформы с самообслуживанием, повышающие коллективный опыт отделов ИТ-эксплуатации и информационной безопасности. Вместо того чтобы постоянно ждать результатов от смежных групп, а потом переделывать заново огромный объем работы, команды трудятся независимо и продуктивно над небольшими заданиями, быстро и часто предоставляя клиентам новые возможности.

Даже релиз широко известных, привлекающих внимание продуктов становится обычным делом, если используются методы «теневого запуска». Задолго до даты запуска код помещается в рабочую среду, но его новые возможности невидимы для всех, кроме персонала компаний-разработчика и небольшой группы реальных клиентов, что позволяет тестировать и развивать новые возможности, пока не будет достигнута поставленная бизнес-цель.

И вместо того чтобы трудиться дни и ночи напролет, пытаясь задействовать новую функциональность, мы просто включаем ее в конфигурационных настройках. Это небольшое изменение делает ее доступной огромному количеству клиентов и предоставляет возможность вернуться назад, если что-либо пойдет не так. В результате релиз нового продукта полностью контролируется, он предсказуем, обратим и не вызывает опасений.

Спокойнее проходят не только те релизы, где добавляются новые функции. На ранних стадиях обнаруживаются и исправляются любые проблемы, пока они не разрослись до катастрофических размеров. Их исправление окажется быстрее и дешевле. С каждым новым исправлением мы добываем для организации новое знание, что позволяет успешно предотвращать проблемы, а впоследствии быстрее обнаруживать и исправлять схожие.

Кроме того, инженеры постоянно обучаются, и при этом формируется организационная культура, основанная на выдвижении гипотез. Научный метод применяется для того, чтобы все подвергалось оценке, а разработка продукта и улучшение процесса разработки расцениваются как эксперименты.

Мы ценим время каждого сотрудника, поэтому не тратим годы на создание функций, не нужных клиентам, не развертываем неработающий код и не исправляем то, что не вызывает проблем.

Будучи нацелены на решение поставленных задач, мы создаем долговременные группы, и они полностью отвечают за достижение нужных результатов. Завершив какой-либо проект, мы не перераспределяем разработчиков по другим командам, в результате чего они лишаются возможности следить за результатами своего труда. Мы сохраняем состав команд. Поэтому их члены могут повторять итерации разработки, улучшая продукт и используя накопившиеся знания, чтобы успешнее достигать поставленных целей. То же самое относится и к командам, создающим продукт для внешних клиентов, так же как внутренние платформенные команды помогают другим командам действовать продуктивнее, успешнее и безопаснее.

Вместо культуры страха мы создаем культуру высокого доверия и сотрудничества. Никто не боится брать на себя риски. Все получают возможность смело обсуждать любые проблемы, не скрывая их и не откладывая на потом. В конце концов, чтобы решить проблему, ее нужно распознать.

И поскольку каждый сам определяет качество своей работы, то он и занимается встраиванием автоматизированного тестирования в свою повседневную деятельность и использует партнерские проверки, чтобы быть уверенным: проблемы будут устранены задолго до того, как окажут воздействие на клиента. Эти процессы снижают риски по сравнению с принятой практикой утверждения кода руководителями, что позволяет предоставлять продукт быстро, надежно и

безопасно и даже убеждать скептически настроенных аудиторов, что у нас есть эффективная система внутреннего контроля.

И если что-то вдруг пойдет не так, мы тщательно анализируем причины неудачи — не чтобы наказать виновного, а чтобы лучше понять, чем вызван сбой и как предотвращать подобное в будущем. Такой ритуал укрепляет культуру обучения. У нас существуют также внутренние конференции, и на них работники могут улучшить навыки: каждый постоянно учит других и учится сам.

Поскольку мы заботимся о качестве, постольку иногда специально создаем сбои в производственной среде, чтобы понять, каким образом система выходит из строя в случаях, которые можно предвидеть. Мы проводим запланированные тренировки по устранению крупномасштабных сбоев, случайным образом «убивая» процессы в производственной системе или выключая серверы, вводим сетевые задержки и делаем другие злонамеренные поступки, чтобы проверить отказоустойчивость. Это позволяет повысить устойчивость системы к сбоям, а также провести обучение персонала и внести улучшения по результатам тестов.

В таком мире каждый человек, какова бы ни была его роль в технологической организации, — хозяин своего труда. Он уверен, что его работа действительно способствует достижению целей компании, что подтверждается низким уровнем сбоев рабочей среды и успехом организации на рынке.

У нас немало убедительных доказательств ценности методов DevOps для бизнеса. В «Докладе о состоянии DevOps» компании Puppet Labs (в создании участвовали Джез Хамбл и Джин Ким) представлены данные за 2013–2016 гг., полученные примерно от 25 000 технических специалистов. Это помогает лучше понять уровень работоспособности и особенности организаций на всех этапах внедрения DevOps.

Прежде всего, эти данные продемонстрировали высокую (по сравнению с прочими) эффективность компаний, использующих DevOps. Сравнивались следующие характеристики:

- показатели пропускной способности;
- развертывание кода и внесение изменений (чаще в 30 раз);
- время, необходимое на разработку кода и внесение изменений в него (в 200 раз меньше);
- показатели надежности;
- развертывание в производство (коэффициент успешности в 60 раз выше);
- среднее время восстановления после сбоя (в 168 раз быстрее);
- показатели эффективности организации;
- производительность, доля на рынке и рентабельность (вероятность улучшить эти показатели в два раза выше);
- рост капитализации рыночной стоимости компаний (за три года на 50 % больше).

Другими словами, более производительные компании оказались и более гибкими, и более надежными. Это наглядное доказательство того, что использование DevOps позволяет выйти из корневого, хронического конфликта. Организации с высокой эффективностью развертывали код в 30 раз чаще, а время, необходимое для перехода от «код зафиксирован» к «успешно работает в реальной среде», было меньше в 200 раз — с периода в несколько недель, месяцев (а порой и квартала) оно сократилось до нескольких минут или часов.

Кроме того, у более производительных компаний имелось в два раза больше шансов повысить прибыльность, долю на рынке и эффективность. А у тех, кто сообщил нам свои биржевые символы, рост капитализации за три года оказался выше на 50 %. Текучка кадров в них ниже, а сотрудники чувствуют удовлетворенность и в 2,2 раза чаще рекомендуют компанию друзьям как отличное место работы. Организации с высокой эффективностью лучше обеспечивают такой показатель, как «информационная безопасность». Достижение целей по безопасности интегрировано во все стадии процессов разработки и эксплуатации, поэтому на исправление соответствующих проблем уходит вдвое меньше времени.

Когда число разработчиков увеличивается, производительность каждого нередко снижается из-за

потери времени на коммуникации, интеграцию и избыточное тестирование. Этот феномен подробно описан в книге Фредерика Брукса «Мифический человеко-месяц, или Как создаются программные системы». В ней он объясняет: когда работа над проектом не выполняется в срок, увеличение количества разработчиков снижает продуктивность не только каждого в отдельности, но и команды в целом.

С другой стороны, DevOps показывает, что при наличии правильной архитектуры проекта, правильных технических методов и правильной производственной культуры небольшие команды разработчиков способны быстро, надежно и независимо от других разрабатывать, выполнять интеграцию, тестировать и развертывать изменения в производственной среде. Согласно наблюдениям Рэнди Шупа, бывшего директора по разработке в компании Google, крупные компании, использующие DevOps, имеют в штате тысячи разработчиков, но их организация и методы позволяют небольшим командам добиваться удивительной продуктивности, как если бы они были стартапом.

В «Докладе о состоянии DevOps в 2015 г.» изучался не только показатель «количество развертываний в день», но и «количество развертываний в день на одного разработчика». Мы выдвинули гипотезу, что высокопроизводительные инженеры могут увеличивать количество развертываний по мере роста численности команды.

Действительно, мы обнаружили такую зависимость. На рис. 1 показано, что при низкой производительности разработчиков количество развертываний в день уменьшается по мере роста численности команды, при средней — остается постоянным, а при высокой — растет пропорционально числу разработчиков.

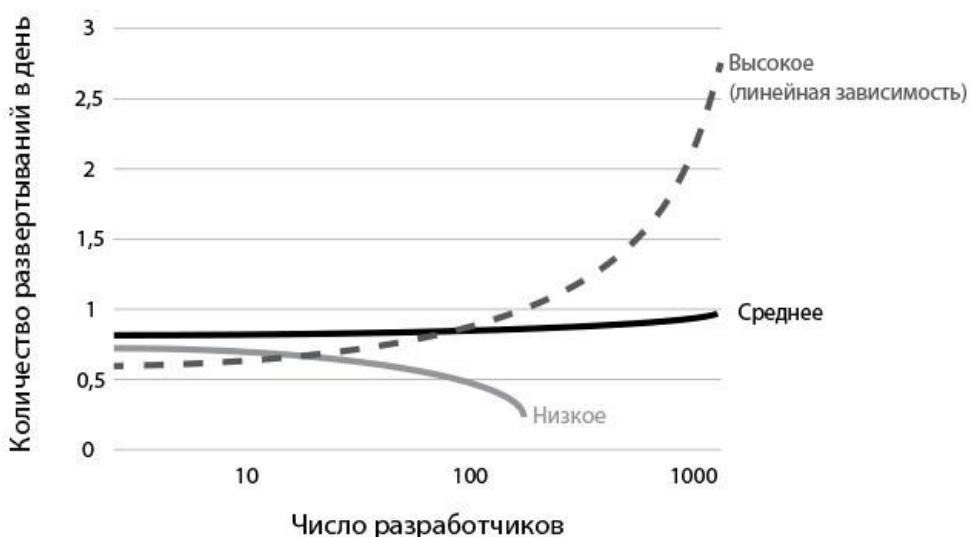


Рис. 1. Количество развертываний в день в зависимости от числа разработчиков (источник: «Доклад о состоянии DevOps в 2015 г.», компания Puppet Labs)

Другими словами, организации, внедрившие DevOps, могут увеличивать количество развертываний в день пропорциональным увеличением числа разработчиков, как это уже сделали компании Google, Amazon и Netflix.

Одна из наиболее авторитетных книг по бережливому производству — «Цель. Процесс непрерывного совершенствования» — написана доктором Элияху Голдраттом в 1984 г. Под ее влияние попало целое поколение руководителей предприятий во всем мире. Это рассказ о директоре завода. Он должен был сократить расходы и наладить выполнение поставок всего за 90 дней, потому что иначе предприятие пришлось бы закрывать.

Позже Голдратт рассказал о письмах, полученных после выхода книги. Обычно их содержание было примерно таким: «Совершенно очевидно, что вы тайно проникли на наш завод, поскольку абсолютно верно описали мою жизнь как директора...» Но гораздо важнее другое: письма показали, что люди способны повторить прорывные действия у себя в организации.

Книга Джина Кима, Кевина Бера и Джорджа Слаффорда «Проект “Феникс”». Роман о том, как DevOps меняет бизнес к лучшему» (год выпуска — 2013-й) очень похожа на «Цель...». Это повесть о руководителе ИТ-организации, столкнувшемся с проблемами, типичными для такого рода компаний: перерасход бюджета, нарушение графика, хотя его соблюдение жизненно важно для компании. Развертывание проекта происходит из рук вон плохо, возникают сложности с доступностью,

безопасностью, соответствием требованиям и так далее. В конечном счете он и его команда начинают использовать методы и принципы DevOps, чтобы справиться с этими проблемами и дать организации возможность выдержать конкуренцию на рынке. Помимо этого, показано, как методы DevOps улучшают рабочую атмосферу в команде, способствуют снижению стресса, повышают удовлетворенность результатами вследствие большей вовлеченности сотрудников в процессы организации в целом.

Как и в случае с «Целью...», в «Проекте “Феникс”...» имеются серьезные доказательства универсальности описанных в книге проблем и решений. Приведем некоторые отзывы, взятые с сайта Amazon: «Я обнаружил сходство между собой и героями этой книги... Вполне возможно, что я встречал таких людей на всем протяжении моей карьеры». «Если вам доводилось работать в любом качестве в IT, эксплуатации или в сфере информационной безопасности, вы, безусловно, почувствуете связь с описанным в этой книге». «В книге “Проект «Феникс»...” нет такого персонажа, которого я не смог бы сравнить с собой или с кем-то, кого знаю в реальной жизни... не говоря уж о проблемах этих персонажей».

В остальных частях этой книги мы опишем, как повторить преобразования, описанные в «Проекте “Феникс”...», и приведем примеры из жизни других организаций, использующих методы и принципы DevOps, чтобы добиться такого же успеха.

Цель данной книги в том, чтобы описать теорию, принципы и методы, необходимые для успешного использования DevOps и достижения желаемых результатов.

В основу этого руководства положены десятилетия изучения теории рационального использования высокопроизводительных технологических организаций. Мы проделали эту работу, помогая компаниям преобразовываться. Наши исследования подтвердили эффективность методов DevOps. Также были использованы материалы бесед с экспертами в соответствующих областях и анализ около ста практических примеров, о которых шла речь на конференции DevOps Enterprise Summit.

Книга разделена на шесть частей, описывающих теорию DevOps, принципы использования «трех путей», особый взгляд на обоснование теории, изложенной в книге «Проект “Феникс”». Роман о том, как DevOps меняет бизнес к лучшему и предназначенный для каждого, кто когда-то занимался технологическими процессами создания продуктов (обычно это понятие включает управление, разработку, тестирование, эксплуатацию и информационную безопасность) или был иным образом вовлечен в указанные процессы. Так же книга предназначена для руководителей бизнеса и маркетологов, ведь именно в этих областях обычно зарождаются технологические инициативы.

Не следует ожидать, что в книге есть глубокий анализ любого из этих направлений, а также DevOps, Agile, ITIL, принципов бережливого управления или описания улучшения процессов. Но каждая из тем затрагивается и разъясняется в других специализированных изданиях, если требуется.

Наша цель — сформировать базовые знания о важнейших концепциях в каждой из областей. Это нужно, чтобы понять основы, а также ввести терминологию и формулировки, необходимые на практике работы для взаимодействия с коллегами, вовлеченными в процесс создания ИТ-ценностей. Ну и для того, чтобы сформулировать общие цели.

Книга окажется полезной руководителям бизнеса и другим заинтересованным лицам, если они полагаются на технологические организации.

Кроме того, книга предназначена тем, кто в своих организациях лишь отчасти сталкивается с описанными здесь проблемами (например, с длительным временем развертывания или со сложным развертыванием, сопряженным с неприятностями). Такие читатели, оказавшиеся в относительно удачном положении, смогут извлечь пользу из понимания принципов DevOps, особенно касающихся общих целей, обратной связи и непрерывного обучения.

В приведен краткий обзор истории DevOps и представлены основы теории и ключевые темы из соответствующих областей знаний, развивавшихся несколько десятилетий. Затем мы представим принципы верхнего уровня — подход «трех путей»: обеспечение полного цикла разработки, петля обратной связи, культура непрерывного обучения и экспериментирования.

В описывается, когда и как надо начинать, представлены такие концепции, как поток создания ценности, принципы и шаблоны организационного проектирования, адаптационные шаблоны, а также приводятся практические примеры.

В описывается, как ускорить процесс разработки через формирование основы для конвейерного развертывания — быстрого и эффективного автоматизированного тестирования, непрерывной интеграции, непрерывной поставки и проектирования низкорискового процесса релизов.

В обсуждаются способы ускорения и усиления обратной связи путем создания эффективной телеметрии для обнаружения, более глубокого анализа и решения проблем, достижения

поставленных целей, формирования обратной связи, позволяющей разработке и эксплуатации безопасно развертывать изменения, встраивать A/B-тестирование в повседневную деятельность и создавать процессы взаимной проверки и координации для повышения качества результатов.

В описывается, как ускорять непрерывное обучение, формируя соответствующую внутреннюю культуру, преобразуя небольшие открытия в глобальные улучшения и правильно резервируя время для организационного обучения и усовершенствований.

И наконец, в мы расскажем, как правильно обеспечивать безопасность и проверять соответствие ее требованиям в повседневной работе, интегрируя превентивный контроль безопасности в общие хранилища исходного кода и услуг, функции обеспечения безопасности — в конвейер развертывания. Необходимо улучшать телеметрию для обеспечения более успешного обнаружения ошибок и восстановления, защиты конвейера развертывания и достижения целей по изменению управления.

Вводя кодификацию этих приемов, мы надеемся ускорить внедрение методов DevOps, повысить их успешность и уменьшить количество энергии, необходимой для запуска преобразований DevOps.

## Введение

В первой части книги «Руководство по DevOps» мы рассмотрим, как слияние нескольких важных тенденций в менеджменте и технологиях создает предпосылки для появления на сцене DevOps. Мы опишем потоки создания ценностей, то, как DevOps становится результатом применения принципов бережливого производства к потокам создания технологических ценностей, а также «три пути»: потоки, обратная связь и постоянное обучение и экспериментирование.

Основные темы части I:

- принцип потока ценности, ускоряющий для клиентов доставку продукта от разработчиков к отделу эксплуатации;
- принцип обратной связи, позволяющий создавать безопасные системы;
- принцип постоянного обучения и экспериментирования, создающий режим наибольшего благоприятствования для организации высокой культуры производства при научном подходе к рискам как составляющей ежедневной работы.

DevOps и связанные с ним технические, архитектурные и культурные методики — результат соединения многих философских и управляемых тенденций. Во многих организациях смогли разработать сходные принципы самостоятельно. Понимание того, что DevOps появился из широкого спектра таких действий, феномен, описанный Джоном Уиллсом (одним из соавторов этой книги) как «конвергенция DevOps», показывает удивительный прогресс мышления и невероятные совпадения. Опыт длиной в несколько десятилетий, полученный при совершенствовании производства, организации управления, пристроенного на высоком уровне доверия, и прочего, привел к тому, что мы сейчас называем методами DevOps.

DevOps — результат применения самых надежных принципов из области лидерства и производства материальных ценностей к потоку создания ценности ИТ. DevOps опирается на базовые понятия теории бережливого производства, теории ограничений, производственной системы компании Toyota, правила создания устойчивых систем, модель обучающихся организаций, культуру психологической безопасности и так далее. Другие важные особенности, которые используют в DevOps, — это управляемая культура высокого доверия, лидерство как служение и управление изменениями в организации. В результате стали возможными высочайшее качество, надежность, стабильность и безопасность, полученные меньшей кровью и за меньшие деньги, увеличение роста и стабильности за счет потока технологической ценности в подразделениях управления продуктом, разработки, тестирования, отделах управления ИТ-эксплуатацией и информационной безопасности.

Хотя можно сказать, что DevOps основывается на принципах бережливого производства, теории ограничений и распространенного в Toyota движения «ката», многие рассматривают его как логическое продолжение Agile-движения, зародившегося в 2001 г.

Такие техники, как «управление созданием потока ценности» и «канбан-доски», были созданы в рамках производственной системы Toyota в 1980-х гг. В 1997 г. Lean Enterprise Institute начал изучать возможность применения принципов бережливого производства к другим вариантам потоков создания ценности, например к сфере обслуживания и здравоохранению.

Два основных принципа бережливого производства заключаются в том, что по *продолжительности производственного цикла*, необходимого для превращения сырья в готовый продукт, можно максимально точно предсказать качество продукта, степень удовлетворенности клиентов и настроение исполнителей. Лучший способ сделать производственный цикл короче — сократить размер каждого производственного задания.

Принципы бережливого производства сфокусированы на том, как с помощью системного мышления создать нечто ценное для клиента за счет постоянства целей, развития научного подхода, потока создания ценности, методов «вытягивания» (pull) (против «заталкивания» (push)), обеспечения высокого качества с первого раза и высокой культуры руководства.

В 2001 г. появился манифест Agile. Его создали 17 авторитетных специалистов в области разработки ПО. Они хотели создать легковесный набор ценностей и принципов в противовес тяжеловесным процессам разработки (например, метод водопада) и методологиям (например, так называемый RUR — рациональный унифицированный процесс).

Один из ключевых принципов манифеста — «эффективно доставлять программное обеспечение часто, раз в две недели или раз в два месяца, делая упор на сокращение сроков». Подчеркивалось стремление к небольшому объему производственных заданий и релизу продуктов с инкрементальными изменениями вместо крупных, «каскадных». Другие принципы подчеркивали

необходимость в небольших самомотивированных командах, действующих в атмосфере высокого уровня доверия.

Считается, что методы Agile способны значительно увеличить продуктивность организаций-разработчиков. И, что интересно, многие из ключевых моментов в истории DevOps соответствуют аналогичным ситуациям, происходившим в сообществе Agile или на конференциях, посвященных этому методу.

В 2008 г. в рамках конференции Agile, проходившей в Торонто, Патрик Дюбуа и Эндрю Шафер провели встречу под названием «Птицы одного полета». Она была посвящена применению принципов Agile к инфраструктуре, а не написанию кода приложений. Поначалу они были единственными приверженцами этой идеи, но быстро обрели единомышленников, включая одного из авторов этой книги — Джона Уиллиса.

Позже, на конференции Velocity в 2009 г., Джон Оллспоу и Пол Хаммонд продемонстрировали новаторскую презентацию под названием «Десять развертываний в день: кооперация разработки и эксплуатации во Flickr». В ней они описали, как создали общие цели для разработчиков (Dev) и эксплуатации (Ops) и использовали методы непрерывной интеграции, чтобы сделать развертывание частью ежедневной работы. Как утверждали впоследствии участники презентации, они сразу же поняли, что присутствуют на историческом мероприятии, имеющем далеко идущие последствия.

Патрик Дюбуа не участвовал в презентации, но был настолько восхищен идеей Оллспоу и Хаммонда, что в том же 2009 г. организовал первую конференцию DevOpsDays в бельгийском Генте, где тогда жил. Так и появился на свет термин DevOps.

Внедряя непрерывную разработку, тестирование и интеграцию, Джез Хамбл и Дэвид Фарлей расширили концепцию *непрерывной поставки*. Это определило важность «конвейера разработки»: код и инфраструктура всегда готовы к развертыванию, весь код прошел проверку и может быть безопасно развернут в производственной среде. Идею впервые представили на конференции Agile в 2006 г. Затем, в 2009 г., Тим Фитц абсолютно независимо придумал то же самое и изложил свои мысли в блоге под заголовком «Непрерывное развертывание».

В 2009 г. Марк Ротер написал книгу «Тойота Ката. Лидерство, менеджмент и развитие сотрудников для достижения выдающихся результатов», где изложил двадцатилетний опыт кодификации производственной системы Toyota. Будучи студентом, он принял участие в поездке руководителей компании General Motors на заводы компании Toyota. Затем ему довелось участвовать в разработке инструментария для внедрения бережливого производства. Он был очень удивлен, что ни одна из компаний, внедривших этот метод, не смогла достичь такого же уровня производительности, как Toyota.

Марк сделал вывод: сообщество Lean упустило из вида наиболее важную часть метода — *улучшение kata*. Он пояснил: каждая организация имеет свои рабочие процедуры, и улучшение kata требует создания структуры для ежедневного, рутинного совершенствования, поскольку повседневный опыт — это и есть то, что повышает результаты. Постоянно действующий цикл, то есть желаемое будущего состояния, еженедельное формулирование целей и непрерывное совершенствование повседневной работы, и составлял основу улучшений в компании Toyota.

Выше мы описали историю DevOps и связанных с ним течений. На их основе и возник метод DevOps. Далее в первой части мы рассмотрим потоки создания ценности, то, как к потокам технологической ценности можно применить принципы бережливого управления и «трех путей», обратной связи и непрерывного обучения и экспериментирования.

## Глава 1. Agile, непрерывная поставка и «три пути»

Здесь представлено введение в основы теории бережливого производства и «трех путей» — из этих принципов сформировалось современное состояние DevOps.

Внимание уделено в первую очередь теории и принципам, выработанным за несколько десятилетий изучения рабочих сред, организации его высокой надежности, создания моделей управления с высоким уровнем доверия, положенных в основу методов DevOps. Получившиеся в результате принципы и схемы действий, а также их практическое применение для создания потока технологической ценности описаны в остальных главах книги.

Одна из фундаментальных концепций бережливого производства — *поток создания ценности*. Сначала мы определим это понятие в рамках промышленности, а затем экстраполируем на DevOps и технологический поток создания ценности.

Карен Мартин и Майк Остерлинг в книге *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation* определили поток создания ценности как «последовательность действий, предпринимаемых организацией с целью выполнить запрос клиента», или как «последовательность действий, необходимых для разработки, выпуска и доставки товаров клиенту, включая оба потока — и информационный, и материальный».

В материальном производстве поток создания ценности легко увидеть, легко наблюдать за ним: он начинается, когда получен заказ от клиента, а сырье для производства поступило на склад. Чтобы обеспечить короткое и предсказуемое время выполнения заказа в любом потоке создания ценности, обычно фокусируются на плавном течении работы, используя такие приемы, как уменьшение размера партии, снижение объема незавершенного производства, недопущение переделок, чтобы исключить попадание дефектных деталей на конечных этапах и постоянно оптимизировать систему для движения в сторону глобальных целей.

Те же принципы и методы, обеспечивающие скорость выполнения работы в процессах материального производства, применимы и к области технологий (и вообще для любой деятельности, создающей знание). В DevOps поток создания технологической ценности обычно определяют как процесс, требующийся для преобразования бизнес-гипотез в технологический сервис, поставляющий ценность заказчику.

Исходные данные для процесса — формулирование бизнес-цели, концепции, идеи или гипотезы. Все начинается, когда мы принимаем задачу в области разработки, добавляя ее к уже имеющимся.

Таким образом, команда разработчиков, следующая типичному Agile- или итеративному процессу, скорее всего, сможет адаптировать эту идею в соответствии с пожеланиями пользователей, создав некоторую разновидность спецификации функциональных возможностей. Затем она реализует их в виде кода приложения или создаваемого сервиса и поместит его в репозиторий системы контроля версий, где каждое изменение интегрируется в основной код и тестируется вместе со всей системой.

Поскольку продукт создается, когда наши сервисы работают в производственной среде, следует обеспечить не только быстрый поток доставки, но и то, чтобы развертывание можно было выполнять без хаоса и перебоев: задержек в обслуживании, нарушения функционирования сервисов, требований безопасности или совместимости.

В остальной части книги сосредоточимся на такой составной части потока создания ценности, как сокращенное время развертывания. Начинается оно тогда, когда некий инженер из команды потока создания ценностей (включающей разработчиков, тестировщиков, отделы эксплуатации и информационной безопасности) получает изменения от системы контроля версий. А заканчивается, когда изменение начинает успешно работать в производственной среде, создавая продукт для клиентов и генерируя обратную связь и телеметрию.

Первый этап включает проектирование и разработку. Он сродни бережливой разработке продуктов и характеризуется высокой изменчивостью и неопределенностью, часто требует творческого подхода к выполнению работы, которая может никогда не понадобиться. Это приводит к различной продолжительности данного этапа. В отличие от него второй этап, включающий тестирование и производство, соответствует принципам бережливого производства. Он требует творческого подхода и опыта и имеет тенденцию к предсказуемости и автоматизации, его цель — обеспечить полезный результат, то есть небольшое предсказуемое время выполнения и близкое к нулю количество дефектов.

Мы не любители больших порций работы, последовательно проходящих через такие части потока создания ценности, как «проектирование и разработка», а затем через поток «тестирование и производство» (например, когда используется большой водопадный процесс или долго живущая функциональная ветвь кода). Напротив, мы стремимся, чтобы тестирование и производство

выполнялись одновременно с проектированием и разработкой. Так обеспечиваются быстрый ход потока и высокое качество. Этот метод успешен, если исполнение осуществляется по небольшим частям и качество обеспечивается на каждом этапе потока создания ценностей.

В Lean время выполнения заказа — одна из двух характеристик, обычно использующихся для измерения производительности потоков ценности. Другая характеристика — время производства (иногда его еще называют временем контактирования или временем выполнения задачи).

Если отсчет времени выполнения заказа начинается в момент оформления и заканчивается при выполнении, то время производства отсчитывается с момента, когда мы начинаем работу над заказом, точнее, не засчитывается тот период времени, когда заказ стоял в очереди на выполнение (рис. 2).



Рис. 2. Время выполнения заказа и время производства

Поскольку время выполнения заказа — самая важная характеристика для клиента, обычная цель — улучшить процессы, сократив вот этот параметр, а не время производства. Однако соотношение времени производства и времени выполнения заказа — важный критерий оценки эффективности: чтобы обеспечить быстрый ход работ и короткое время выполнения заказа, почти всегда требуется сократить время ожидания, пока дойдет очередь.

При ведении бизнеса обычными способами мы часто оказываемся в ситуации, когда развертывание занимает несколько месяцев. Особенно часто это происходит в больших сложных организациях, использующих тесно связанные друг с другом монолитные приложения, плохо интегрированные в среду для тестирования, со значительной продолжительностью тестирования и длительным временем развертывания в рабочей среде, высокой зависимостью от тестирования вручную и необходимостью одобрения многочисленными инстанциями в компании. Когда такое происходит, поток создания ценности начинает выглядеть, как показано на рис. 3.

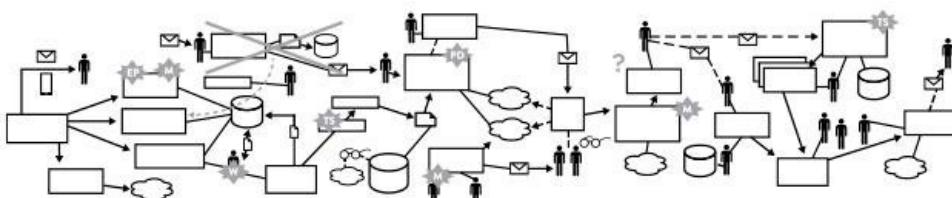


Рис. 3. Поток технологической ценности при продолжительности внедрения длиной в три месяца (пример взят из вышедшей в 2015 г. книги Дэвона Эдвардса DevOps Kaizen)

При большой продолжительности развертывания сверхусилия требуются практически на каждой стадии потока создания ценности. Может оказаться, что при завершении проекта, когда все результаты труда инженеров собраны воедино, все перестанет работать, код не будет собираться правильно или перестанет проходить тесты. Исправление любой проблемы, определение, кто именно «сломал» код и как это исправить, потребует нескольких дней или даже недель, а в результате отдача для клиентов окажется невысокой.

В идеальном случае при работе с DevOps разработчики постоянно быстро получают обратную связь, что дает им возможность быстро и независимо внедрять, интегрировать и валидировать код, а также обеспечивать развертывание кода в производственной среде (это могут делать как они сами, так и другой отдел).

Это достигается за счет постоянной проверки небольших изменений в коде, производимых в

репозитории системы контроля версий, выполнения автоматического и пред-производственного тестирования изменений, а затем развертывания в реальной производственной среде. Что и позволяет нам быть твердо уверенными: сделанные изменения после развертывания будут функционировать так, как задумано, и любая возникшая проблема будет быстро обнаружена и исправлена.

Наиболее легко это достигается, когда архитектура модульная, хорошо инкапсулированная, в ней отсутствуют тесные связи между компонентами, так что небольшие группы имеют возможность работать с высокой степенью автономности, возникающие сбои оказываются небольшими и с ограниченными последствиями, не вызывающими глобальных нарушений работоспособности системы.

При таком сценарии время развертывания измеряется минутами или в худшем случае часами. Получившаяся карта потока ценности должна выглядеть примерно так, как показано на рис. 4.



Рис. 4. Технологический поток создания ценности с развертыванием за минуты

Помимо времени выполнения заказа и времени производства для оценки технологического потока создания ценности применяется и третий показатель — доля завершенной и правильной работы (percent complete and accurate — %C/A). Этот показатель отражает качество выполнения на каждом этапе потока создания ценности. Карен Мартин и Майк Остерлинг утверждают: «показатель %C/A можно получить, опросив клиентов, как часто в количественном выражении они получали продукт, пригодный к использованию сразу. Подразумевается, что они используют результат без необходимости его корректировать, добавлять пропущенную информацию или пояснения к имеющейся, если она недостаточно понятна».

В книге The Phoenix Project «три пути» представлены как основополагающие принципы. Из них выводится DevOps и его методы (рис. 5).

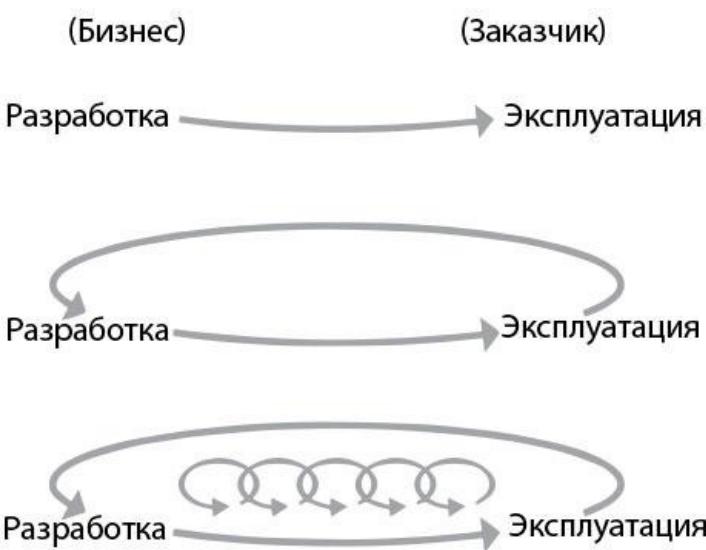


Рис. 5. «Три пути» (пример взят из текста Джина Кима The Three Ways: The Principles Underpinning DevOps, размещенного в блоге Revolution Press blog по адресу: , доступ осуществлен 9 августа 2016 г.)

Первый путь обеспечивает быстрое течение потока слева направо от разработки к эксплуатации, а затем к клиентам. Чтобы сделать это течение как можно более интенсивным, необходимо сделать результаты видимыми, уменьшая размеры заданий и интервалы между ними, обеспечивая качество

путем предотвращения попадания дефектов на конечные этапы и постоянно занимаясь оптимизацией для достижения глобальной цели организации.

За счет ускорения прохождения работы через поток создания технологической ценности мы сокращаем время для выполнения запросов внутренних или внешних клиентов, особенно необходимое для развертывания кода в производственной среде. При этом повышается качество и результативность, а также способность превзойти конкурентов.

Получившиеся в результате методы включают непрерывную разработку, интеграцию и тестирование, а также процессы развертывания: создание различных сред по требованию, ограничение параллельно исполняемых задач и построение систем и организаций, допускающих изменения без риска.

«Второй путь» обеспечивает быстрый и постоянный поток обратной связи справа налево на всех этапах потока создания ценности. Это требует от нас усиления обратной связи с целью предотвратить повторное появление проблем и получить более быстрое обнаружение и восстановление. При этом мы обеспечиваем качество уже на исходном этапе и создаем или встраиваем знание в те этапы, где оно необходимо, — это позволяет нам создавать все более безопасные системы: проблемы обнаруживаются и исправляются задолго до того, как может произойти катастрофический сбой.

Если распознавать проблемы сразу после их появления и решать их, то тем самым можно постоянно укорачивать петли обратной связи, а это один из основных постулатов всех без исключения современных методологий совершенствования процессов. Это делает максимально перспективными возможности нашей организации учиться и улучшаться.

Третий путь позволяет создать продуктивную культуру высокого доверия. Она поддерживает динамический, упорядоченный и научный подход к экспериментам и рискованным решениям, содействует извлечению уроков как из успехов организации, так и из ее неудач. Кроме того, постоянно сокращая петлю обратной связи, мы создаем все более безопасные системы и имеем возможность активнее принимать рискованные решения и проводить эксперименты, помогающие учиться быстрее конкурентов и выигрывать на рынке.

В рамках третьего пути мы также проектируем наши работы таким образом, чтобы мы могли увеличить результативность нового знания, преобразовать открытия, сделанные одним из отделов, в глобальные улучшения. Таким образом, независимо от того, где инженеры выполняют задания, они делают это с учетом накопленного коллективного опыта всей организации.

В этой главе мы рассказали о концепции потоков создания ценности, о времени развертывания как одной из ключевых мер эффективности производственного и технологического потоков ценности и о концепциях высокого уровня, за каждой из которых стоят «три пути» — принципы, лежащие в основе DevOps.

В следующих главах «три пути» рассматриваются подробно. Первый из этих принципов — поток. Здесь главное обеспечить быстрое выполнение заказа в любом потоке создания ценности — как материальных, так и технологических. Методы, обеспечивающие быстрый поток работы, описаны в третьей части.

## **Глава 2. Первый путь: принципы потока**

В потоке создания технологической ценности работа обычно течет от отдела разработки к эксплуатации, в функциональных областях, находящихся между бизнесом и клиентами. Первый путь требует быстрого и плавного протекания потока задач между этими отделами с целью быстрее доставить продукт клиентам. Оптимизация идет для этой глобальной цели, а не для локальных, например показателей завершения разработки, критериев соотношения поиска и исправления ошибок или степени готовности ИТ-отдела.

Ход процессов можно ускорить, делая работу прозрачной благодаря уменьшению размеров заданий и интервалов между ними, повышая качество путем предотвращения попадания дефектов на конечные этапы производства. Ускоряя протекание процессов с помощью потока создания технологической ценности, мы сокращаем время выполнения заказов, поступающих от внешних или внутренних клиентов, повышая качество, что делает нас более гибкими и дает возможность превзойти конкурентов.

Наша цель — уменьшить затраты времени, необходимые на развертывание изменений в рабочей среде, и улучшить стабильность и качество продукта. Подсказки, как это сделать для технологического потока ценности, можно найти в принципах бережливого производства, примененных к потоку создания ценностей в материальном производстве.

Существенная разница между технологическим и материальным потоком создания ценности в том, что наша работа невидима. В отличие от физических процессов в технологическом потоке создания ценности нельзя увидеть воочию задержки в движении или то, как работа скапливается перед перегруженными участками. Перенос материальной деятельности на другой участок обычно заметен, хотя и происходит медленно, поскольку необходимо выполнить физическое перемещение деталей.

А чтобы переместить на другой участок технологическую часть, достаточно нажать кнопку или щелкнуть мышкой, например передав выполнение задания другой команде. Поскольку делается это легко и просто, задания порой могут без конца «скакать» между командами из-за недостатка информации. Или же работа может быть передана на завершающие этапы с дефектами, незаметными до тех пор, пока разворачивание не будет выполнено в последний момент или же приложение не начнет сбоить в производственной среде.

Чтобы видеть, где осуществляется движение, а где очередь или простой, необходимо сделать процесс как можно более прозрачным. Один из лучших методов — использование визуализации наподобие доски канбан или доски планирования спринтов, где можно представить ход выполнения с помощью картонных или электронных карточек. Результат появляется с левого края (нередко ее вытягивают из списка невыполненных дел), передается с одного рабочего места к другому (они представлены колонками), и путь его завершается, когда результат достигает правого края доски: обычно это колонка, подписанная «сделано» или «в производстве».

При этом деятельность не только становится прозрачной: ею можно управлять, чтобы она как можно быстрее передвинулась с левого края к правому. Больше того, можно измерить время выполнения, то есть время, за которое карточка, попав на доску, перешла в колонку «сделано».

В идеале на доске канбан должен быть показан весь поток создания ценности, чтобы работа обозначалась как завершенная при достижении правого края доски (рис. 6). Процесс не завершается, когда разработчики заканчивают реализацию функций. Нет, это происходит только тогда, когда приложение успешно действует в производственной среде, поставляя продукт клиентам.

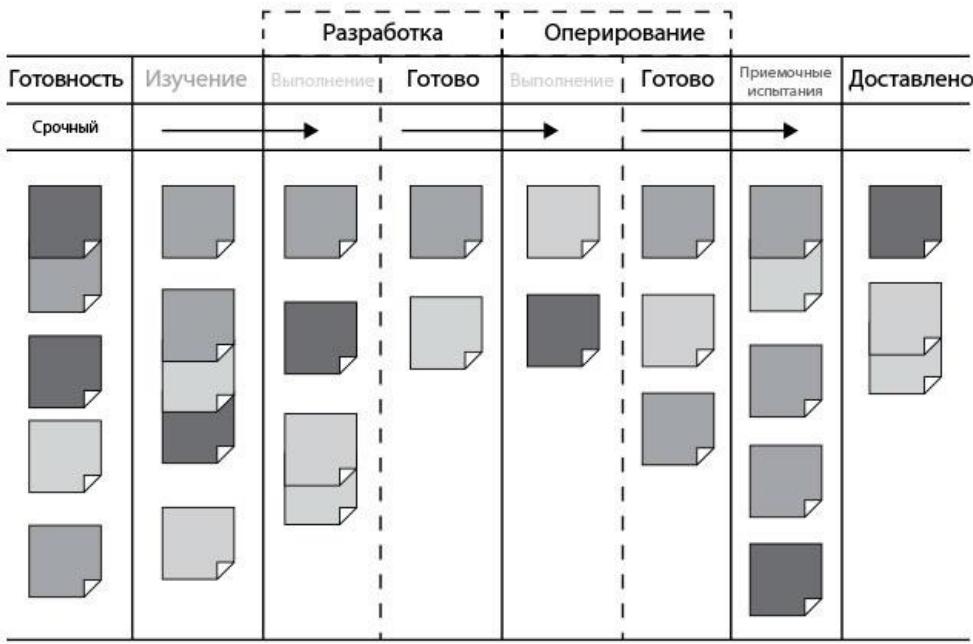


Рис. 6. Пример доски канбан, охватывающей формулирование требований, разработку, тестирование, подготовку к производству и позицию «в производстве» (источник: Дэвид Андерсон и Доминика Деграндис, учебные материалы для бизнес-тренинга Kanban for ITops, 2012)

Если все рабочие задания для каждого рабочего места размещены в очереди и эта очередь вместе с заданиями прозрачна, то все участники проекта могут определить свои приоритеты в контексте глобальных целей. Это позволяет сотрудникам на каждом рабочем месте, увеличивая собственную отдачу, сосредоточиваться на одной задаче, наиболее приоритетной с точки зрения завершения общего процесса.

В материальном производстве ежедневный цикл обычно продиктован производственным графиком. Он составляется с определенной периодичностью (например, ежедневно или еженедельно). В нем указывается, что должно быть сделано для выполнения клиентского заказа и к какому сроку, какие детали должны быть изготовлены и так далее.

В технологическом производстве работа оказывается гораздо более динамичной, особенно в случае совместно используемых сервисов, когда команды должны удовлетворить требования множества разных заинтересованных лиц. В результате приоритетом становится не повседневная деятельность, а то, «как надо сделать сегодня»: требование выполнить все незамедлительно, поступающее по всем возможным каналам коммуникаций, включая систему отслеживания ошибок, аварийные вызовы, сообщения по электронной почте, телефонные звонки, сообщения в чате и от менеджеров.

Нарушения процесса в материальном производстве также хорошо заметны и дорого стоят, нередко они требуют прекратить выполнение текущего задания, выбросить незавершенные результаты и запустить новое задание. Трудоемкость всех этих усилий заметно уменьшает стремление правильно отрабатывать сбои, если они случаются часто.

Напротив, прервать процесс в технологическом производстве легко, поскольку последствий практически никто не видит, хотя отрицательное влияние на продуктивность может оказаться намного сильнее, чем в материальном производстве. Например, инженер, назначенный сразу на несколько проектов, должен переключаться между задачами, а это каждый раз влечет дополнительные затраты на формирование контекста деятельности, на восстановление в памяти правил и целей конкретной задачи.

Исследования показали: время завершения даже простых действий, например по расстановке геометрических фигур, значительно снижается при многозадачном режиме. И, конечно, поскольку взаимодействие с технологическим потоком ценности гораздо более сложно в когнитивном плане, чем сортировка геометрических фигур, последствия деятельности в многозадачном режиме сильнееказываются на времени выполнения.

Деятельность в многозадачном режиме можно ограничить, использовав для управления доску канбан, например, путем кодификации незавершенной работы (НзП — незавершенное производство) и установки максимального размера для каждой колонки или каждого рабочего места. Это ограничит количество карточек, одновременно находящихся в одной колонке.

Скажем, можно установить лимит НзП на тестирование в три карточки. И если в тестовой колонке уже есть три карточки, то нельзя добавить еще одну, пока одна из имеющихся не будет завершена или пока карточка не будет удалена из колонки «выполнение» и помещена обратно в очередь (то есть вернется в самую левую колонку). Никакую деятельность нельзя начать выполнять, пока на столе не появится карточка с задачей. Именно это гарантирует, что можно видеть все.

Доминика Деграндис, один из ведущих экспертов по использованию канбан в потоке создания ценности DevOps, заметила: «Управление размером очереди НзП — мощный инструмент, поскольку это один из немногих показателей времени выполнения заказа, а ведь в большинстве случаев мы не можем сказать, сколько времени займет та или иная операция, пока она не будет сделана».

Ограничение НзП также облегчает обнаружение проблем, препятствующих завершению. Например, установив такое ограничение, мы обнаруживаем, что в данный момент нам нечего делать, поскольку мы ждем, что финальную точку поставит кто-то другой. Хотя на первый взгляд кажется заманчивым начать делать нечто новое (действуя по принципу «лучше хоть что-то, чем безделье»), гораздо полезнее понять, что именно приводит к задержкам, и помочь решить эту проблему. Дурная многозадачность зачастую появляется, когда инженеров назначают сразу на несколько проектов, в результате чего возникают проблемы с определением приоритетов.

Другими словами, как язвительно заметил Дэвид Андерсон, автор книги *Kanban: Successful Evolutionary Change for Your Technology Business*, «заканчивайте начинать, начинайте заканчивать».

Другой ключевой компонент для создания гладкого и быстрого потока — выполнение задач небольшими порциями. До революции, произведенной идеей бережливого производства, обычной практикой считалось изготовление больших партий, особенно в случаях, когда переключение с одного вида продукции на другой требовало много времени или было дорогостоящим. Например, для изготовления крупных деталей кузовов автомобилей требуется установка на штамповочные машины больших и тяжелых штампов, и этот процесс может занять несколько дней. Когда переключение на другую продукцию настолько дорого стоит, часто стараются изготовить как можно больше деталей за раз, чтобы сократить число переналадок.

Однако большие размеры партий приводят к тому, что стремительно растет размер НзП и повышается нестабильность всех элементов производственной цепочки. В результате время выполнения заказов сильно увеличивается, а качество заметно ухудшается — если в детали обнаружен дефект, то в металлом отправляется вся партия.

Один из ключевых уроков бережливого производства в том, что для сокращения времени выполнения заказа и повышения качества необходимо постоянно уменьшать размеры партий. Теоретический нижний предел размера — *поштучное изготовление*, когда каждая операция производится над одной единицей продукции.

Огромную разницу между большим и малым размерами партии можно увидеть на примере симуляции рассылки газеты, описанной в книге Джеймса Вумека и Дэниела Джонса «Бережливое производство. Как избавиться от потерь и добиться процветания вашей компании».

Предположим, что в этом примере мы отправляем по почте десять буклетов и на каждый требуется четыре действия: сложить бумагу, вложить ее в конверт, заклеить конверт, поставить на конверте штамп.

Стратегия больших партий (то есть массовое производство) заключается в том, чтобы по очереди выполнять каждую из этих операций с десятью буклетами. Другими словами, сначала складывают десять листов бумаги, затем каждый из них помещают в конверт, заклеивают все десять конвертов и, наконец, проштампывают их.

С другой стороны, при мелкосерийной стратегии (то есть поштучном потоке) все шаги, необходимые для отправки каждого буклета, выполняются последовательно, а затем действия повторяются с новым буклетом. Другими словами, сначала мы складываем листы одного экземпляра, вкладываем их в конверт, заклеиваем его и ставим штамп — только после этого можно переходить к следующему буклету.

Разница между использованием большого и малого размеров партии значительна (рис. 7). Предположим, каждая из четырех операций занимает десять секунд для каждого из десяти конвертов. При большом размере партии первый заклеенный и проштампованный конверт будет готов только через 310 секунд.

## Большие партии



## Партия единичного размера



Рис. 7. Симуляция «игры в письма» (сложить, вложить, заклеить, проштамповывать) (источник: Стефан Люйтэн, запись «Поток с единичным размером партии: почему массовое производство — не самый эффективный способ что-то сделать» в блоге от 8 августа 2014 г., )

Еще хуже вот что: если на этапе заклеивания конверта окажется, что при складывании допущена ошибка, то она обнаружится только через 200 секунд после начала, и все десять буклетов надо будет заново сложить и опять поместить в конверты.

В отличие от этого при мелкосерийном производстве первый готовый конверт будет сделан через сорок секунд, в восемь раз быстрее по сравнению со стратегией больших партий. И если на первом шаге была допущена ошибка, то она распространится только на один буклет. Малый размер партии ведет к меньшему НзП, сокращению срока выполнения заказа, более быстрому обнаружению ошибок и минимизации количества переделок.

Плохие результаты, связанные с большим объемом партии, применимы к потоку создания технологической ценности в той же мере, что и к производственному. Представим, что существует годовой график выпуска ПО. Согласно ему весь годовой объем кода, написанного разработчиками, выпускается в производственную среду.

Так же как и при производстве материальных ценностей, большой размер партии может внезапно создать высокий уровень НзП, массовые нарушения деятельности на рабочих местах нижнего уровня и обусловить низкое качество результатов. Это подтверждает общий опыт: чем больше изменений в производственных процессах, тем труднее выявляются и устраняются производственные ошибки и тем дольше происходит восстановление производства.

В своем блоге Startup Lessons Learned Эрик Райс утверждает: «Размер партии — это количество продукции, передающееся как единое целое с одного этапа на другой в ходе процесса разработки (или DevOps). В случае программного обеспечения самый простой вариант видимого пакета — код. Каждый раз, когда инженер загружает написанный код в систему контроля версий, создается партия определенного объема. Существует множество методов контроля за этими партиями, начиная от крошечных пакетов, необходимых для непрерывного развертывания, до более традиционных методов разработки на основе ветвления кода, когда весь код, написанный многими разработчиками, увязывается в один пакет и из него составляют единое целое».

В технологическом потоке создания ценности эквивалент поштучного производства реализуется непрерывным развертыванием, когда каждое изменение, сделанное в системе контроля версий, интегрировано, протестировано и развернуто в производство. Методы, позволяющие сделать это, описаны в четвертой части.

В технологическом потоке создания ценности длительное время развертывания, измеряемое месяцами, нередко обусловлено тем, что для переноса кода из системы контроля версий в производственную среду зачастую требуются сотни (или даже тысячи) операций. Для передачи кода по потоку создания ценности необходимо, чтобы несколько отделов решали множество задач, осуществляя и поддерживая функциональное тестирование, интеграционное тестирование, создание рабочей среды, администрирование серверов, администрирование систем хранения данных, работу сети, балансировку нагрузки и информационную безопасность.

Каждый раз, когда задание передается от одной команды к другой, необходимы разного рода коммуникации: запросы, уточнения, уведомления, действия по координации и нередко приоритизации, планирование, разрешение конфликтов, тестирование и верификация. Это может потребовать использования различных систем учета ошибок или системы управления проектами, написания технических документов и спецификаций, общения на совещаниях, или посредством сообщений электронной почты, или телефонных звонков, с помощью общего доступа к файлам, FTP-серверов и вики-страниц.

Каждый из этих шагов создает потенциальную возможность появления очереди, и работа будет ждать до тех пор, пока мы не получим возможность использовать ресурсы, распределяемые между различными потоками создания ценности (например, в случае централизованной эксплуатации). Время выполнения таких запросов нередко оказывается настолько велико, что для корректировки и соблюдения сроков необходимо постоянное вмешательство руководителей более высокого уровня.

Даже в самых благоприятных условиях некоторая часть информации неизбежно теряется при каждой передаче эстафеты. При большом количестве передач от одного отдела к другому работа может полностью утратить контекст выполнения или даже потерять связь с целями организации. Например, системный администратор может обнаружить созданное обращение с просьбой о создании учетной записи пользователя. В нем не будет сведений о том, почему ее нужно создать, с какими приложениями или службами она создается, какие для нее должны быть установлены зависимости. Может быть, это повторение уже выполненного задания.

Для предотвращения проблем такого типа необходимо стремиться сократить количество случаев передачи работы, либо автоматизировать значительную ее часть, либо реорганизовать команды, с тем чтобы они могли заняться предоставлением ценности клиенту, а не разбираться с постоянной зависимостью от других. В результате мы увеличиваем поток создания ценности, снижая расход времени на простой и исключая период, когда добавленная стоимость не создается (см. ).

Чтобы уменьшить время выполнения заказа и увеличить отдачу, необходимо постоянно выявлять затруднения, возникающие в системе, и увеличивать ее производительность. В уже упоминавшейся книге «Цель. Процесс непрерывного совершенствования» Голдратт утверждает: «В любом потоке создания ценности всегда существует направление этого потока и обязательно есть один-единственный сдерживающий фактор: любое улучшение, не влияющее на этот фактор, иллюзорно». Если мы можем улучшить работу на рабочем месте, расположенному перед препятствием, то она будет накапливаться в узком месте еще быстрее, и это вызовет ожидание других сотрудников.

С другой стороны, если мы сможем улучшить функционирование рабочего места, расположенного за «бутылочным горлышком», то сотрудник все равно будет простоявать, ожидая, пока объект пройдет через узкое место. В качестве решения Голдратт предложил «пять шагов сосредоточения на проблеме»:

- обнаружить затруднения в работе системы;
- определить, что следует улучшить в месте затруднения;
- подчинить все остальные задачи решению проблемы;
- сделать ограничения, накладываемые проблемой, более мягкими;
- если предыдущие шаги оказались неудачными — возвратиться к первому шагу, но не позволять бездеятельности нарушить всю систему.

Во время типичных преобразований, выполняемых при переходе к DevOps, когда время развертывания сокращается с месяцев (или даже кварталов) до минут, затруднения обычно развиваются в следующей последовательности.

- **Создание среды:** нельзя добиться развертывания по требованию, если постоянно приходится ждать несколько недель и даже месяцев создания производственной среды или среды для тестирования. Контрмера — создание сред по первому требованию на самостоятельное обслуживание, чтобы они всегда были доступны в тот момент, когда в них возникает необходимость.
- **Развертывание кода:** нельзя добиться развертывания по требованию, если каждое из развертываний работающего кода занимает несколько недель или месяцев (например, для каждого развертывания требуется 1300 операций, проведенных вручную и, следовательно, подверженных ошибкам, требующих участия не менее 300 инженеров). Контрмера — максимальная автоматизация развертывания, чтобы оно стало полностью автоматизированным и его мог выполнить любой разработчик.
- **Тестовые настройка и запуск:** нельзя добиться развертывания по требованию, если каждое развертывание кода требует не менее двух недель на настройку тестовых сред и наборов данных и еще до четырех недель для выполнения вручную всех регрессионных тестов. Контрмера — автоматизация тестов: так можно безопасно выполнять развертывания параллельно с тестированием, соотнося быстроту тестирования кода со скоростью разработки.

- **Чрезмерно жесткая архитектура:** нельзя добиться развертывания по требованию, если чрезмерно жесткая архитектура подразумевает, что каждый раз, когда мы хотим изменить код, приходится отправлять инженеров для участия в десятках заседаний различных комитетов, чтобы они могли получить разрешение на внесение этих изменений. Контрмера — создание более слабосвязанной архитектуры, чтобы можно было бы вносить изменения безопасно и с большей автономией, увеличивая тем самым производительность труда разработчиков.

Когда все эти трудности будут преодолены, возникнет следующее препятствие — группа разработчиков или владельцы продукта. Поскольку цель — дать возможность небольшим группам разработчиков самостоятельно разрабатывать, тестировать, а также быстро и надежно развертывать продукты для клиентов, то именно здесь можно ожидать затруднений. Где бы ни трудились ведущие инженеры — в отделе разработки, тестирования, эксплуатации или информационной безопасности, — они имеют одну цель: помочь максимально повысить продуктивность разработчиков.

Когда появляются затруднения, единственное, что мешает их преодолеть, — недостаток хороших бизнес-идей и возможности создания кода, необходимого для проверки гипотез с участием реальных клиентов.

Приведенная выше последовательность затруднений — обобщение типичных методов преобразований с целью выявить препятствия в реальных потоках создания ценности, например через создание карты потока ценностей и его измерение. Такие методы описаны ниже в этой книге.

Шигео Шинго, один из пионеров создания производственной системы Toyota, считает: потери — наибольшая угроза жизнеспособности бизнеса. В бережливом производстве для этого часто используется определение «использование любых материалов или ресурсов сверх того, что клиент требует и за что готов платить». Шинго определил семь основных типов потерь на производстве: излишние запасы, перепроизводство, лишние этапы обработки, ненужная транспортировка, ожидание, потери из-за ненужных перемещений и потери из-за выпуска дефектной продукции (брата).

В более современных интерпретациях бережливого производства отмечалось, что термин «исключить потери» может иметь уничтожительную окраску. Вместо этого цель реструктурирована, чтобы уменьшить трудности и тяготы в повседневной работе за счет непрерывного обучения для достижения целей организации. В дальнейшем слово «потери» будет использоваться в более современном смысле, поскольку оно лучше соответствует идеалам DevOps и желаемым результатам.

В своей книге «Бережливое производство программного обеспечения. От идеи до прибыли» Мэри и Том Поппендиц описывают потери и задержки в потоке разработки программного обеспечения как то, что вызывает задержки у клиента, — например, действия, которые можно не выполнять без ущерба для результата.

Перечисленные ниже категории потерь и задержек взяты из упомянутой книги, если не оговорено иное.

- **Работа, выполненная частично:** она включает в себя и незавершенные в потоке создания ценности задачи (например, документы с требованиями или распоряжения о внесении изменения еще не рассмотрены), и находящиеся в очереди (например, ожидание отчета тестировщиков или ответа системного администратора на запрос). Частично сделанное устаревает и со временем теряет ценность.
- **Излишняя обработка:** любые дополнительные задачи, выполняемые в рамках процесса, но не добавляющие ценности для клиента. Это может включать в себя написание документации, не используемой на рабочих местах нижнего уровня, или обзоров и утверждений, не добавляющих результирующей ценности. Излишняя обработка требует дополнительных усилий и увеличивает время.
- **Излишняя функциональность:** «фичи», встроенные в продукт, но не востребованные ни самой организацией, ни клиентами (так называемые блестяшки или украшательства). Излишние функции преумножают сложность продукта и усилия, требующиеся для тестирования и управления функциональностью.
- **Переключение между задачами:** когда сотрудник задействован в нескольких проектах и потоках создания ценности, необходимость переключаться между различными контекстами и зависимостями требует дополнительных усилий и затрат времени.

- **Ожидание:** любые задержки, требующие ресурсов: приходится ждать, пока они не освободятся. Задержки увеличивают время цикла и не дают клиенту получать ценность.
- **Лишние движения:** количество усилий, чтобы переместить информацию или материалы с одного рабочего места на другое. Лишние движения могут появиться, когда люди, нуждаются в частом общении, располагаются далеко друг от друга. Задержки также могут вызывать лишние движения и часто требуют дополнительных коммуникаций для устранения неясностей.
- **Дефекты:** неправильная, отсутствующая или неясная информация, неподходящие материалы или продукты создают потери, поскольку необходимы значительные усилия для решения этих вопросов. Чем больше времени проходит между появлением дефекта и его обнаружением, тем труднее устранить дефект.
- **Ненормированная или ручная работа:** расчет на ненормированную или проведенную вручную работу, которую должны сделать другие, — например, использование серверов, сред тестирования и конфигураций без функции восстановления. В идеале любые зависимости от отдела эксплуатации должны быть автоматизированы, находиться на самообслуживании и быть доступны по требованию.
- **Геройство:** чтобы организация могла достичь своих целей, отдельные лица или группы вынуждены порой выполнять чрезвычайные действия. Они могут даже стать частью повседневной деятельности (например, устранение проблем с производством, возникших в два часа ночи, создание сотен заявок на поддержку как обычную составляющую часть каждого выпуска ПО в производство).

Наша цель — сделать прозрачными затруднения и потери везде, где требуется геройство, и систематически осуществлять все необходимое для их смягчения или устранения, чтобы достичь цели — ускорить поток создания ценности.

Улучшение процесса протекания технологического потока создания ценности много значит для получения результатов с помощью DevOps. Это достигается за счет того, что деятельность становится прозрачной, ограничивается НзП, уменьшаются размер партии и количество передач от сотрудника к сотруднику. Затем, в ходе повседневной работы, они устраняются.

Конкретные рекомендации, обеспечивающие быстрое течение потока создания ценностей DevOps, будут даны в четвертой части. В следующей главе мы расскажем о втором пути — принципах обратной связи.

### **Глава 3. Второй путь: принципы обратной связи**

Первый путь — принципы, обеспечивающие быстрое протекание потока создания ценности слева направо. Второй путь включает принципы, позволяющие обеспечить быстрый и непрерывный поток обратной связи в противоположную сторону, справа налево, на всех этапах потока создания ценности. Цель — создать более безопасную и более устойчивую систему.

Это особенно важно при работе в сложных системах, где необходимо использовать первую же возможность, чтобы обнаружить и исправить ошибки, обычно тогда, когда возможны катастрофические последствия — производственная травма или активизация атомного реактора.

В технологических отраслях мы действуем почти исключительно внутри сложных систем с высоким риском катастрофических последствий. Как и в материальном производстве, мы часто обнаруживаем проблемы только при больших неудачах, таких как массовое производство неработоспособной продукции или нарушение безопасности в результате кражи данных клиента.

Мы делаем систему безопаснее, создавая быстрый, интенсивный, высококачественный поток информации через нашу организацию на протяжении всего пути создания ценности. Эта система включает в себя петлю как обратной, так и прямой связи. Такой подход позволяет обнаруживать и устранять проблемы, пока они еще небольшие и их дешевле и проще исправлять, не допуская катастрофы, проводить организационное обучение, интегрируемое в будущую деятельность. При возникновении сбоев или аварий мы рассматриваем их как возможности для обучения, а не занимаемся поиском виновных.

Но давайте вначале изучим характер сложных систем и то, каким образом их можно сделать безопасными.

Вот одна из определяющих характеристик сложной системы: она требует от любого человека увидеть целое и понять, как в нем соединяются все фрагменты. Сложные системы обычно имеют высокую степень взаимозависимости тесно связанных компонентов, и системный уровень нельзя понять лишь с точки зрения поведения компонентов системы.

Доктор Чарльз Перроу изучал аварию на АЭС Three Mile Island и отметил: никто не сумел бы предположить, как реактор поведет себя во всех обстоятельствах, каким образом он может выйти из строя. Проблема скрывалась в одном элементе, который было сложно отделить от других, и быстро и непредсказуемо распространялась.

Доктор Сидни Деккер, занимавшийся, в частности, кодифицированием некоторых ключевых элементов культуры безопасности, заметил еще одну характерную черту сложных систем: при повторении одних и тех же действий повторный результат может оказаться непредсказуемым, или, иначе говоря, повторение не обязательно приведет к тем же самым результатам. Именно эта особенность делает списки проверок и наилучшие практики, остающиеся неизменными в течение долгого времени, недостаточными для предотвращения критических последствий (см. ).

Поэтому, поскольку сбои неизбежны в сложных системах, необходимо спроектировать безопасную систему, будь то в материальном производстве или в технологическом. Сделать возможной работу без опасений и с уверенностью, что любые ошибки будут обнаружены быстро, задолго до того, как они станут причиной серьезных последствий: травм исполнителей, дефектов продукции или отрицательного воздействия на клиента.

Доктор Стивен Спир, защитивший в Гарвардской школе бизнеса диссертацию, посвященную расшифровке механизма производственной системы Toyota, заявил: разработка абсолютно безопасных систем лежит, скорее всего, за пределами наших способностей, но мы можем сделать сложные системы более безопасными при выполнении следующих четырех условий:

- сложная работа управляется так, чтобы проблемы, возникающие при разработке и эксплуатации, было возможно обнаружить;
- проблем множество, они решаются, и в результате быстро накапливаются новые знания;
- знания, полученные в одном из подразделений, используются во всей организации;
- лидеры готовят других лидеров, и возможности организации постоянно увеличиваются.

Каждое из этих условий необходимо для безопасной работы в сложной системе. В следующих разделах описаны первые два условия и их значение, а также как они были созданы в других

областях и каковы методы включения их в технологический поток создания ценности (третье и четвертое условия описаны в ).

При безопасной системе работы необходимо постоянно проверять проектное решение и эксплуатационные допущения. Цель — увеличить поток информации в системе как можно раньше, быстрее, дешевле и с прозрачной взаимосвязью между причинами и следствиями. Чем больше предположений мы сможем проверить, тем быстрее сумеем найти и устранить проблемы, увеличить устойчивость, гибкость и способность к обучению и инновациям.

Мы можем сделать это, создав петли обратной и прямой связи в системе работы. Доктор Питер Сенге в книге «Пятая дисциплина. Искусство и практика самообучающейся организации» описал петли обратной связи как чрезвычайно важную часть процесса обучения мышлению в категориях организаций и систем. Петли обратной и прямой связи дают возможность компонентам системы взаимно усиливать или нейтрализовать друг друга.

В материальном производстве отсутствие эффективной обратной связи часто влияет на основные качества товара и проблемы безопасности. В одном подробно описанном случае на заводе General Motors во Фримонте не существовало эффективных процедур для выявления проблем во время процесса сборки, не было четкого описания, что делать, если проблемы обнаружатся. В результате бывали случаи установки двигателей задом наперед, выпуска машин без рулевого колеса или шин, бывало даже, что автомобили приходилось буксировать со сборочной линии, поскольку они не заводились.

В отличие от этого в высокорезультативных производствах отложен быстрый, частый и высококачественный поток информации на протяжении всего потока создания ценности: каждая рабочая операция измеряется и контролируется, любые дефекты или значительные отклонения быстро обнаруживаются, затем принимаются соответствующие меры. Это основа обеспечения высокого качества, безопасности, непрерывного обучения и совершенствования.

В технологическом потоке создания ценности мы нередко получаем невысокие результаты из-за отсутствия быстрой обратной связи. Например, в каскадном подходе разработки программного обеспечения мы можем писать код целый год и не получать при этом обратную связь о качестве до тех пор, пока не начнется этап тестирования. Или, что еще хуже, пока программное обеспечение не будет передано клиентам.

Когда обратная связь приходит поздно и редко, выясняется, что уже невозможно предотвратить нежелательные последствия.

В отличие от этого цель — создать быструю обратную и прямую связь, когда работа выполняется, на всех этапах технологического потока создания ценности, включающих управление продуктом, разработку, тестирование, информационную безопасность и эксплуатацию. Это подразумевает автоматизированные процессы сборки, интеграции и тестирования, чтобы можно было немедленно обнаружить ситуацию, когда внесенное изменение нарушает правильное функционирование продукта и делает его непригодным к развертыванию.

Мы также создаем всепроникающую телеметрию, чтобы видеть, что все компоненты системы работают в производственной среде, чтобы быстро обнаружить ситуации, когда функционирование нарушается. Телеметрия также позволяет оценить, достигаем ли мы поставленных целей. В идеале отслеживается весь поток создания ценности, так что мы можем видеть, как наши действия влияют на другие части системы и систему в целом.

Петли обратной связи не только обеспечивают быстрое обнаружение и исправление проблем, но также дают информацию, как предотвратить эти проблемы в будущем. Это повышает качество и безопасность системы и создает возможность организационного обучения.

Как сказала Элизабет Хендрисон, технический директор компании Pivotal Software и автор книги *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*, «когда я возглавляла подразделение тестирования, я описывала свою работу как “создание циклов обратной связи”». Обратная связь — важнейший фактор, поскольку она позволяет управлять разработкой. Мы должны постоянно сверять нужды клиентов с нашими стремлениями и тем, что у нас получается. Тестирование — это лишь одна из форм обратной связи».

Очевидно, недостаточно просто обнаружить, что происходит нечто непредвиденное. При возникновении проблемы мы должны объединиться вокруг нее, мобилизовав всех, кто требуется для решения этой проблемы.

Согласно Спиру, цель такого объединения — ограничить проявление проблем, прежде чем они широко распространятся, диагностировать и решить их, чтобы они не смогли появиться снова. «Поступая так, — говорит он, — мы создаем глубокое знание того, как управлять системами, чтобы они делали нашу работу, превращая неизбежное имеющееся вначале незнание в знание».

Пример — шнур Toyota Andon (далее в качестве равнозначного будет использоваться термин «шнур-андон»). На заводах Toyota на каждом рабочем месте натянут сигнальный шнур, всех работников и менеджеров учат дергать за него, когда что-то выходит из строя, например деталь имеет дефект, нужная деталь отсутствует или работа занимает больше времени, чем положено по графику.

Когда кто-то дергает шнур-андон, руководитель команды или подразделения получает сигнал тревоги и немедленно начинает работать над устранением проблемы. Если ее не удается решить за определенное время (например, за 55 секунд), то конвейер останавливается и весь цех приходит на помощь, чтобы решить проблему, пока не будет разработано эффективное средство ее преодоления.

Вместо того чтобы ходить вокруг да около или планировать поиск решения проблемы на тот момент, «когда у нас будет больше времени», мы объединяемся, чтобы исправить ситуацию немедленно, — это практически полная противоположность описанной выше ситуации на заводе General Motors во Фримонте. Объединиться, решая проблему, необходимо по следующим причинам.

- Это предотвращает такое развитие событий, когда усилия, требуемые на решение проблемы, и стоимость решения растут по экспоненте и накапливаются технический долг.
- Это предотвращает начало выполнения новых заданий на рабочих местах, а эти новые задания могут внести в деятельность системы новые ошибки.
- Если проблема не будет решена, то на следующей операции (например, спустя 55 секунд) могут возникнуть те же проблемы на тех же рабочих местах. Однако они потребуют больше исправлений и дополнительной работы (см. ).

Подобную практику можно противопоставить общепринятой, поскольку мы намеренно позволяем локальным проблемам нарушить общий ход выполнения операции. Однако объединение вокруг решения проблемы дает возможность проводить обучение. Это предотвращает потерю критически важной информации из-за забывчивости или изменения обстоятельств. Это особенно важно в сложных системах, где многие проблемы возникают по причине неожиданного и прихотливого сплетения взаимодействия людей, процессов, продукции, места и обстоятельств. И через некоторое время практически невозможно определить, что же именно произошло, когда возникла проблема.

Как отмечает Спир, объединиться вокруг проблемы — это часть программы «учиться распознавать проблемы в реальном времени, определять их причины... и лечить (принимать контрмеры или корректировать производственный процесс). Это обычная практика цикла Стюарта (планирование — действие — проверка — корректировка), популяризированного Эдвардсом Демингом, но форсированного до сверхзвуковой скорости».

Исправить ситуацию до того, как произойдет катастрофа, можно, только объединившись вокруг небольшой проблемы, обнаруженной на раннем этапе. Другими словами, если атомный реактор начал плавиться, уже поздно пытаться избежать неблагоприятных последствий.

Чтобы обеспечить быструю обратную связь в технологическом потоке создания ценности, мы должны создать эквивалент шнура-андон и соответствующей обратной связи. Это требует также формирования производственной культуры, делающей использование шнура неопасным для того, кто за него потянул. Наоборот, поощряется сигнал, что нечто выходит из строя, будь то сбой в производстве или обнаружение ошибки, возникшей на предыдущих этапах, например, если кто-то вносит изменение, нарушающее непрерывный процесс сборки или тестирования.

Когда срабатывает триггер вышеупомянутого шнура-андон, мы собираемся вместе, чтобы решить проблему и предотвратить переход к новой фазе, пока сбой не будет устранен. Это обеспечивает быструю обратную связь для всех, кто задействован в потоке создания ценности (особенно для работника, повинного в сбое), позволяет быстро локализовать и диагностировать проблему, предотвращает дальнейшее накопление усложняющих факторов, скрывающих причину и следствие.

Предотвращая переход к новой фазе, мы осуществляем непрерывную интеграцию и развертывание — единый процесс в технологическом потоке создания ценности. Все изменения, проходящие непрерывную проверку сборки и интеграции, развертываются в производство, а любые изменения, заставляющие наши тесты «дернуть за шнур-андон», объединяют вокруг себя работников.

Мы можем поневоле закрепить небезопасную систему, если не будем активно реагировать на аварии и происшествия. В сложных системах добавить проверок и этапов утверждения — значит увеличить и вероятность будущих сбоев. Полезность процессов утверждения уменьшается, если мы принимаем решение не там, где выполняем проект. Это не только снижает качество, но и увеличивает время, и ослабляет обратную связь между причиной и следствием, и уменьшает нашу способность извлекать уроки из успехов и неудач.

Подобное можно наблюдать даже в небольших и не очень сложных системах. Обычно иерархическая бюрократическая система управления неэффективна, когда несовпадение того, «кто должен это сделать», и того, «кто в действительности это делает», воздействует слишком сильно из-за недостаточной прозрачности и несвоевременности действий.

В качестве примеров неэффективного контроля качества можно привести следующие ситуации:

- требование к другой команде — выполнение трудоемких, подверженных ошибкам и исполняемых вручную задач, хотя их легко автоматизировать и запускать по мере необходимости, когда первая команда нуждается в выполненной работе;
- требуется утверждение результата другим человеком, занятым другими задачами и находящимся далеко от места выполнения работы, что вынуждает его принимать недостаточно компетентные решения или просто автоматически завизировать присланный документ;
- создание больших объемов документации с ненужными подробностями, устаревающими практически сразу после того, как записаны;
- раздача больших объемов работы в группы и специальные комитеты для утверждения и обработки и затем длительное ожидание ответа.

Вместо этого нужно, чтобы каждый человек, занятый в потоке создания ценности, искал и исправлял проблемы в своей зоне ответственности в рамках повседневной работы. Тем самым мы передаем исполнителю ответственность за качество и безопасность труда, чтобы работа фактически выполнялась, а не полагаемся на утверждение документов руководителями, находящимися на отдалении.

Мы используем взаимные проверки предлагаемых изменений, чтобы быть уверенными: изменения будут осуществляться как задумано. Мы автоматизируем максимальную часть проверок качества, обычно выполняемых тестировщиками или отделом информационной безопасности. Вместо того чтобы разработчики отправляли запрос на тестирование или ставили его в свой план, такие тесты выполняются по требованию, что позволяет разработчикам быстро проверить код и даже самостоятельно развернуть изменения в производственную среду.

При этом мы побуждаем каждого исполнителя, а не целое подразделение, отвечать за качество. Информационная безопасность — не просто работа отдела информационной безопасности, так же как доступность — компетенция не только отдела эксплуатации.

Если разработчики разделяют ответственность за качество систем, то они не только улучшают результаты, но и ускоряют процесс обучения. Это особенно важно для разработчиков, обычно наиболее удаленной от клиента группы. Гэри Грувер отмечает: «Разработчикам невозможно научиться чему-либо, когда на них кричат за то, что они сломали шесть месяцев тому назад, — именно поэтому нам необходимо обеспечить обратную связь для всех и как можно скорее, в течение минут, а не месяцев».

В 1980-е годы принципы проектирования для производства подразумевали разработку деталей и процессов таким образом, чтобы законченные изделия имели минимальную стоимость, максимальное качество и малое время изготовления. В качестве примера приводились чрезмерно асимметричные детали — их нельзя было установить неправильно, и проектирование шуруповертов — с их помощью невозможно слишком сильно затянуть гайки.

Это было отклонением от обычных правил конструирования. В них основное внимание уделяется внешним клиентам, а интересами изготавителей пренебрегают.

Бережливое производство определяет два типа клиентов, для которых нужно выполнять конструирование: внешний клиент (вероятнее всего, он платит за поставляемые услуги) и внутренний (получает и обрабатывает задание сразу же после нас). В соответствии с правилами бережливого производства наиболее важный клиент — это наш смежник: к нему ведет поток создания ценности. Оптимизация работы требует, чтобы мы вникали в его проблемы в целях более эффективного выявления проблем проектирования, мешающих быстрому и беспрепятственному течению потока.

В технологическом потоке создания ценности мы выполняем оптимизацию в интересах рабочих нижнего уровня потока, разрабатывая процесс, в котором операционные нефункциональные требования (например, архитектура, производительность, стабильность, testируемость, конфигурируемость и безопасность) приоритетны так же высоко, как и пользовательские функции.

При этом мы формируем качество с самого начала, что с высокой вероятностью выразится в создании набора кодифицированных нефункциональных требований. Затем мы можем рационально интегрировать их в каждую созданную службу.

Создание быстрой обратной связи имеет важнейшее значение для достижения качества, надежности и безопасности в технологическом потоке создания ценности. Мы делаем это, обнаруживая проблемы по мере возникновения, объединяясь вокруг них и добывая новые знания, обеспечивая качество с начальных этапов и постоянно выполняя оптимизацию в интересах рабочих мест нижнего уровня.

Конкретные рекомендации, обеспечивающие быстрое течение потока создания ценности DevOps, представлены в . В следующей главе мы расскажем о третьем пути — принципах обратной связи.

## **Глава 4. Третий путь: принципы непрерывного обучения и экспериментирования**

В то время как первый путь описывает ход рабочего процесса слева направо, а второй — быструю и постоянно действующую обратную связь справа налево, третий путь протекает в постоянном обучении и экспериментировании. Для отдельных работников становится возможным постоянно создавать знания и превращать их из индивидуальных в общие, для команд и организаций в целом.

В материальном производстве, где сохраняются систематические сложности с качеством и безопасностью, работы обычно жестко распределены и проводятся строго в соответствии с установленными правилами. Например, на заводе GM во Фримонте, описанном в предыдущих главах, сотрудники имели очень мало возможностей для применения улучшений и приобретенных знаний в повседневной деятельности, а все предложения что-либо улучшить «упирались в каменную стену безразличия».

В таких производственных средах зачастую существует атмосфера страха и недоверия: рабочих, допускающих ошибки, наказывают, а тех, кто предлагает усовершенствования или указывает на недостатки, рассматривают как провокаторов и смутьянов. В подобных случаях руководители активно подавляют обучение и совершенствование и даже наказывают за них, надолго закрепляя имеющиеся проблемы с качеством и безопасностью.

Высокопроизводительные производства, наоборот, требуют активного содействия обучению вместо жестко заданных заданий, а система работы должна быть динамической, чтобы рабочие, занятые на основном производстве, могли экспериментировать в своей повседневной деятельности. Это становится возможным благодаря тщательной стандартизации процедур работы и документирования результатов.

В технологическом потоке создания ценности наша цель — формирование культуры высокого доверия, подкрепляющей стремление учиться всю жизнь, в том числе методом проб и ошибок, в ходе повседневной работы. Применяя научный подход к процессу внедрения улучшений, и к разработке продуктов, мы можем извлекать уроки из успехов и неудач, выявляя неплодотворные идеи и совершенствуя продуктивные. Более того, любое знание, полученное на одном рабочем месте, можно быстро сделать всеобщим, чтобы новые методы и практики могли быть использованы в масштабах всей организации.

Мы выделяем время, чтобы оптимизировать повседневную работу и в дальнейшем обеспечить и ускорить обучение сотрудников. Мы непрерывно вносим дополнительную нагрузку в системы, чтобы активизировать их постоянное совершенствование. Мы даже имитируем сбои в производственных процессах, правда, в контролируемых условиях, чтобы проверить надежность результатов.

Благодаря созданию динамичной системы непрерывного обучения, помогающей выиграть в рыночной конкурентной среде, мы даем возможность группам быстро адаптироваться к постоянно меняющимся условиям.

Работая в сложной системе, по определению невозможно точно предсказать все результаты любого действия. Это ведет к неожиданным или даже катастрофическим последствиям и несчастным случаям даже тогда, когда мы принимаем меры предосторожности и делаем все тщательно.

Когда эти несчастные случаи сказываются на наших клиентах, мы стремимся понять, почему это произошло. Главной причиной зачастую считается человеческий фактор. До сих пор наиболее частая реакция руководства — попытка пристыдить сотрудника («позор тебе, NN»), чья деятельность вызвала проблему. Подспудно или явно, но менеджмент тем самым намекает: тот, кто виновен в ошибке, будет наказан. Затем создаются дополнительные процессы работы и узаконивается необходимость получать одобрение от вышестоящего руководства, чтобы предотвратить повторное появление похожей ошибки.

Автор термина «просто культура» доктор Сидни Деккер, кодифицировавший некоторые ключевые элементы культуры безопасности, писал: «Реакция на происшествия и аварии, кажущаяся несправедливой, может препятствовать безопасности расследования, стимулировать появление страха вместо понимания ситуации у тех, кто выполняет действительно важную для безопасности работу. Организация еще более бюрократизируется, вместо того чтобы повышать заботу о работниках, и культивирует секретность, уклонение от ответственности и стремление к самозащите».

Эти вопросы особенно остро встают в процессе технологического потока создания ценности. В этом случае работа практически всегда выполняется в сложных системах, и если руководство, реагируя на сбои и происшествия, будет культивировать атмосферу страха, то маловероятно, что кто-то вообще решится сообщить об ошибке. В результате проблема останется скрытой, пока не приведет к катастрофическим последствиям.

Доктор Рон Веструм одним из первых увидел важность организационной культуры в соблюдении техники безопасности и обеспечении производительности. Он отметил, что в медицинских организациях «генеративные» культуры — один из основных прогностических факторов безопасности пациентов. Он определил три типа культуры.

- Для патологических организаций характерен высокий уровень страха и угроз. Сотрудники часто не делятся друг с другом информацией, утаивают ее по причинам внутрикорпоративной политики или искажают ее, стремясь выглядеть лучше. Неудачи зачастую скрываются.
- Бюрократические организации характеризуются жесткими правилами и процессами, зачастую каждое подразделение поддерживает на своей «поляне» собственные правила. Неудачи проходят через систему разбирательств, после чего выносится решение наказать или простить.
- Генеративные организации характеризуются активным поиском и распространением информации, каким образом лучше выполнить свою задачу. Ответственность распределяется по всему потоку создания ценности, а неудачи ведут к размышлению и поиску истинной причины.

<b>Патологическая</b>	<b>Бюрократическая</b>	<b>Генеративная</b>
Информация скрывается	Информация может быть проигнорирована	Информация активно добывается
Известившие о неполадках «нейтрализованы»	Известивших терпят	Известивших обучают
Ответственность перекладывается	Ответственность делится	Ответственность общая
Связи между командами не рекомендуются	Связи между командами разрешены, но не рекомендуются	Связи между командами поощряются
Неудачи скрываются	Организация справедлива и милосердна	Причины неудач изучаются
Новые идеи подавляются	Новые идеи создают проблемы	Новые идеи приветствуются

Рис. 8. Созданная Веструмом модель организационной топологии: как организации обрабатывают информацию (источник: Рон Веструм. Типология организационной культуры // BMJ Quality & Safety 13, no. 2 (2004), doi:)

Так же как в исследованных Веструмом организациях здравоохранения, генеративная культура высокого доверия определяет информационную и организационную производительность в технологическом потоке создания ценности.

Мы формируем основы генеративной культуры, стремясь создать безопасную систему работы. Когда происходят авария или сбой, мы, вместо того чтобы найти человека, допустившего ошибку, ищем способ перестроить систему, чтобы избежать повторения подобных случаев.

Например, мы можем проводить безупречный анализ причин неудачи после каждого инцидента. Цель — понять, как произошла авария, и прийти к согласию по поводу оптимальных контрмер для оптимизации системы, максимально быстрого обнаружения и исправления проблемы и предотвращения рецидива.

Делая это, мы порождаем организационное обучение. Так, Бетани Макри, инженер в Etsy, возглавляющая создание инструмента Morgue, помогающего вскрыть и записать события для анализа причин сбоя, заявила: «Исключив ответственность, вы устраняете страх; устранив страх, допускаете честность; тогда честность дает возможность предотвратить сбой».

Доктор Спир наблюдал: когда прекращается осуждение и начинается организационное обучение,

«организации начинают как никогда активно ставить себе диагнозы и самосовершенствоваться, гораздо лучше обнаруживать проблемы и решать их».

Многие из этих признаков были также описаны доктором Питером Сенге как атрибуты обучающихся организаций. В книге «Пятая дисциплина. Искусство и практика самообучающейся организации» он писал, что эти характеристики помогают клиентам, обеспечивают качество, создают конкурентные преимущества, делают работников более энергичными и заинтересованными и обнажают правду.

Команды часто не могут или не желают повысить эффективность процессов, в рамках которых действуют. В результате они не только продолжают страдать от проблем: с течением времени неприятности только усиливаются. Майк Ротер заметил в уже упоминавшейся книге «Тойота Ката...», что в отсутствие улучшений процессы не остаются прежними — из-за хаоса и энтропии они с течением времени действительно идут только хуже.

Когда в технологическом потоке создания ценности мы избегаем исправлять проблемы, полагаясь на ежедневно применяемые методы обойти их, они накапливаются, равно как и технический долг. Потом выяснится: все свои усилия мы тратим на попытки обойти проблему, пытаясь избежать неприятностей, и у нас уже не остается времени для продуктивной работы. Вот почему Майк Орзен, автор книги Lean IT, заметил: «Важнее повседневной работы — улучшение повседневной работы».

Мы можем оптимизировать повседневную работу, целенаправленно выделяя время, чтобы сокращать технический долг, устранять дефекты и выполнять рефакторинг, наводя порядок на проблемных участках кода и рабочей среды. Мы можем сделать это, резервируя циклы на каждом интервале развертывания или составляя график kaizen blitz (молниеносных улучшений), то есть периодов, когда инженеры самоорганизуются в группы для работы над исправлением любой проблемы по выбору.

В результате каждый постоянно ищет и устраниет проблемы в своей области, и это часть его повседневной работы. Решив наконец повседневные вопросы, замалчиваемые целые месяцы или даже годы, мы сможем перейти к искоренению менее очевидных проблем. Обнаруживая все более слабые сигналы о сбоях и исправляя ошибки, мы тем самым устраним проблемы не только тогда, когда это легче и дешевле, но и когда последствия еще невелики.

Рассмотрим следующий пример — повышение безопасности на рабочем месте в Alcoa, компании по производству алюминия, получившей в 1987 г. 7,8 миллиарда долларов дохода. Производство алюминия требует чрезвычайно высокой температуры, высокого давления и использования агрессивных химикатов. В том же году компания установила пугающий рекорд: 2 % от 90 тысяч сотрудников получили травмы — семь пострадавших в день. Когда Пол О'Нил стал CEO, его первой целью стало исключить травматизм среди сотрудников, подрядчиков и посетителей.

О'Нил хотел иметь возможность не позднее чем через 24 часа узнавать о получении травмы на производстве — не для того, чтобы наказывать, а чтобы убедиться: урок из этого случая извлечен, выводы сделаны, меры по созданию более безопасного рабочего места приняты. В течение десяти лет количество производственных травм в Alcoa сократилось на 95 %.

Сокращение производственного травматизма позволило Alcoa сосредоточить внимание на небольших сложностях и слабых сигналах о неисправностях. Вместо того чтобы посыпать О'Нилу уведомления о произошедших несчастных случаях, его стали информировать и о предотвращенных. Поступая таким образом, руководство компании улучшало безопасность рабочих мест на 20 лет вперед. В результате получались завидные отчеты о безопасности.

Как пишет Спир, «сотрудники Alcoa перестали придумывать обходные пути, чтобы справиться с трудностями, неудобствами и препятствиями. Преодоление трудностей, срочные исправления и решения были постепенно вытеснены во всей организации динамическим определением возможностей улучшить рабочие процессы и конечные продукты. Когда эти возможности определены, а проблемы исследованы, накопившиеся в компании запасы неведения и пренебрежения преображаются в зерна знания». Это помогло компании Alcoa получить конкурентные преимущества на рынке.

Точно так же мы делаем систему работы более безопасной в технологическом потоке создания ценности. Это происходит по мере того, как мы находим и устранием проблемы со все более слабыми сигналами о неисправности. Например, мы первоначально можем проводить расследование только случаев, когда пострадали клиенты. Со временем мы можем перейти к случаям, затрагивающим отдельные команды, и даже к ошибкам, еще не успевшим вызвать сбои.

Когда на рабочем месте или в группе делаются локальные выводы, необходимо также, чтобы существовали механизмы распространения полученного знания на всю организацию, использования этого знания и извлечения из него выгоды. Другими словами, если группа лиц или отдельный работник имеют опыт, повышающий их компетентность, мы обязаны превратить

несистематизированное знание (его трудно передать другому лицу в письменном или устном виде) в явное, кодифицированное, способное через практику применения стать компетентностью другого человека.

Возникает гарантия, что когда кто-либо еще возьмется за аналогичную работу, он станет использовать коллективный опыт тех, кто ранее занимался такой же работой. Замечательный пример превращения локальных знаний в глобальные — программа ВМС США по разработке атомных двигательных систем (также известная как NR — Naval Reactors, военно-морские реакторы). В ней более 5700 реакторо-лет работы без единого несчастного случая, связанного с поражением радиацией.

Программа NR известна сильной приверженностью к сценариям выполняемых процедур и стандартизированной работе, а также строгой необходимости отчетов обо всех происшествиях в случае отхода от процедуры или обычных действий. Это делается, чтобы накапливать знания, причем независимо от того, насколько незначителен сигнал о сбое — процедуры постоянно обновляются и системы конструируются на основе сделанных выводов.

В результате, когда новый экипаж впервые выходит в море, команда извлекает пользу из коллективного знания, полученного из 5700 реакторо-лет безаварийной работы. Не менее впечатляет то, что и собственный опыт, накопленный в открытом море, будет добавлен в коллективное знание, поможет будущим экипажам безопасно выполнять задания.

В технологическом потоке создания ценности мы должны создать аналогичные механизмы для формирования глобальных знаний — от стремления сделать анализ произошедших неприятностей доступными для выполнения в них поиска всеми командами, решающими подобные проблемы, и до создания общих хранилищ исходных кодов, охватывающих всю организацию. Общий код, библиотеки и описания конфигураций впитывают ценнейшие коллективные знания всей организации и могут быть использованы. Все эти механизмы помогают преобразовать индивидуальную компетентность в знания, принадлежащие всем в организации.

Организации, занятые материальным производством и имеющие невысокую производительность, во многих отношениях тем самым защищают себя от перебоев — другими словами, они всегда могут ускориться или нарастить «подкожный жирок». Например, чтобы уменьшить рискостоя на рабочих местах (из-за позднего прибытия сырья, отбраковки, сделанной на складе, и т. п.), менеджеры могут создать на каждом рабочем месте больший запас заготовок. Однако такой буферный запас увеличивает НЗП, что ведет к различным нежелательным последствиям, описанным выше.

Точно так же, чтобы снизить рискостоя рабочих мест из-за неисправностей оборудования, менеджеры могут увеличить производственные мощности, купив больше оборудования, наняв больше людей и даже увеличив площадь производственных помещений. Но все эти меры приведут к увеличению расходов.

И наоборот, передовые работники могут добиться тех же или даже более высоких результатов, улучшая повседневную работу, непрерывно отыскивая узкие места, чтобы повысить производительность, а также устойчивость работы производственной системы.

Рассмотрим типичный эксперимент на одном из предприятий компании Aisin Seiki, одного из ведущих поставщиков сидений для компании Toyota. Предположим, что на нем есть две производственные линии. Каждая способна производить сто единиц продукции в день. В дни небольшой загрузки они могут выполнять заказы на одной из линий, а на другой проводить эксперименты по повышению производительности и поиску уязвимых мест в рабочем процессе, зная, что если на первой линии произойдет сбой, то производство можно будет перенести на вторую.

Непрестанным экспериментированием в повседневной работе они смогли увеличить мощность производства, часто без добавления нового оборудования или найма дополнительных работников. Сложившиеся в результате этих улучшений шаблоны работы повысили не только устойчивость, но и производительность труда, поскольку организация всегда находится в состоянии напряженности и изменений. Процесс применения стресса с целью повышения устойчивости был назван придумавшим его риск-аналитиком Нассимом Талебом antifragility (антихрупкость).

В технологическом потоке создания ценности мы можем ввести в системы такой же элемент напряженности, постоянно стремясь снизить затраты времени на развертывание, увеличить охват тестированием, уменьшить время выполнения тестов и даже изменить архитектуру, если это необходимо для роста продуктивности работы разработчиков или увеличения надежности.

Мы также можем провести день учений (игровой день), отработав действия при крупномасштабных отказах, например отключении всех центров обработки данных. Или можем внести в производственную среду еще более серьезные неисправности (например, с помощью программы

Chaos Monkey, созданной в компании Netflix: она в случайном порядке убивает процессы или нарушает работу серверов в производстве), чтобы убедиться, что система действительно устойчива настолько, насколько мы хотим.

Традиционно ожидается, что лидеры будут отвечать за формирование целей, выделение ресурсов для достижения этих целей и установление правильного сочетания стимулов. Лидеры также создают эмоциональную атмосферу в своей организации. Другими словами, ведут за собой, принимая правильные решения.

Однако в настоящее время существуют веские доказательства того, что руководитель не может достичь авторитета только за счет принятия правильных решений. Он должен создавать условия, чтобы его команда достигла максимума в повседневной работе. Другими словами, результат требует усилий и руководителей, и работников, и они взаимно зависимы.

Джим Вумек, автор книги *Gemba Walks for Service Excellence: The Step-by-Step Guide for Identifying Service Delighters*, описал необходимые взаимодополняющие рабочие отношения и взаимное уважение между лидерами и рядовыми работниками. Согласно Вумеку, взаимосвязь необходима, поскольку ни одна из сторон не может решить проблемы в одиночку: лидеры недостаточно близки к рабочим местам, хотя это может быть необходимо для решения проблем, а рядовые работники не обладают широтой взгляда на работу организации в целом и не имеют права вносить изменения за пределами своей компетенции.

Лидеры должны повысить значение обучения и упорядочить способы устранения неисправностей. Майк Ротер formalизовал эти методы, назвав их *coaching kata* (ката наставничества). В результате получились методы, отражающие научный подход. Мы можем четко выразить свои истинные цели, например «поддержание нулевого числа аварий» в случае Alcoa или «удвоение производительности за год» в случае Aisin.

Эти стратегические цели обуславливают формирование итеративных и более краткосрочных, идущих каскадом и затем выполняющихся путем установления условий на уровне потока создания ценности или рабочего центра (например, «сокращают время выполнения работ на 10 % в течение следующих двух недель»).

Целевые условия задают рамки научного эксперимента: мы ясно формулируем проблему для решения, строим предположения, как предлагаемые нами контрмеры помогут ее снять, разрабатываем методы тестирования предположений, истолковываем полученные результаты и используем полученные знания как основу для следующей итерации.

Лидер помогает обучать работника, проводящего эксперимент, задавая ему вопросы. Например, такие.

- Каким был ваш последний шаг и что получилось?
- Что вам удалось узнать?
- Каково состояние проблемы сейчас?
- Какая цель будет у вашего следующего шага?
- Над преодолением какого препятствия вы сейчас работаете?
- Каким будет ваш следующий шаг?
- Какого результата вы ожидаете?
- Когда мы можем его проверить?

При таком подходе лидеры помогают работникам увидеть и решить повседневные проблемы. Недаром это ключевой элемент производственной системы Toyota, организации обучения, улучшений Ката и высокой надежности работы компании. Майк Ротер отметил, что видит компанию Toyota «организацией, характеризующейся в первую очередь уникальными поведенческими процедурами, обеспечивающими постоянное обучение всех членов».

В технологическом потоке создания ценности научный подход и итеративный метод направляют процессы внутренних улучшений. Мы проводим эксперименты, чтобы убедиться: создаваемые нами продукты действительно помогут внутренним и внешним клиентам в достижении их целей.

Принципы третьего пути удовлетворяют потребности в оценке организационного обучения,

обеспечивая высокую доверительность и взаимное перекрытие между функциями, признавая, что в сложных системах сбои всегда будут иметь место, и делая приемлемым обсуждение проблем, с тем чтобы мы могли создать безопасную систему работы. Это также требует институционализации улучшений повседневной работы, преобразования локальных знаний во всеобщие. Их можно использовать в рамках всей организации. Также неплохо вводить в работу элемент напряженности.

Хотя формирование культуры непрерывного обучения и экспериментов — основа третьего пути, оно также вплетено в первый и второй. Другими словами, улучшение потока и обратной связи требует итеративного и научного подхода, включающего формирование граничных условий целевого состояния, формирования гипотез, помогающих разработать и провести эксперименты, и оценки результатов.

Результатом будет не только лучшая производительность, но также повысившаяся устойчивость, более высокая доверенность работой и повышенная адаптивность организации.

В первой части книги мы сделали обзор нескольких положений, сыгравших роль при создании DevOps. Мы также рассмотрели три основных принципа, формирующих основу для успешного использования DevOps в организациях: принципы потока, обратной связи, непрерывного обучения, экспериментирования. Во второй части мы выясним, как начать внедрять движение DevOps в вашей организации.

## **Введение**

Как решить, в каком подразделении организации начать запуск DevOps-преобразований? Кто должен в этом участвовать? Как организовать команды, поддерживая их работоспособность и максимально увеличивая шансы на успех? Именно на эти вопросы мы ответим в части II книги «Руководство по DevOps».

В следующих главах мы изучим процесс инициирования DevOps-преобразований. Начнем с оценки потоков создания ценности в организации, определим подходящее место для начала преобразований и будем формировать стратегию для создания команды, выделенной для преобразований, с учетом конкретных целей совершенствования и последующего расширения области работы. Для каждого потока создания ценности определим необходимую работу и затем рассмотрим организационную разработку стратегий и организационные архетипы, наилучшим образом обеспечивающие достижение целей преобразования.

Основное внимание в этих главах уделяется:

- выбору, с какого потока создания ценности начать;
- необходимости понимать, какая именно работа выполняется в нашем кандидате на поток создания ценности;
- проектированию организации и архитектуры с учетом закона Конвея;
- созданию решений, благоприятных для выхода на рынок, путем более эффективного сотрудничества в рамках потока создания ценности;
- защите команд и созданию условий для их работы.

Начало любых преобразований характеризуется полной неопределенностью: мы намечаем путь к идеальному конечному состоянию, но практически все промежуточные шаги неизвестны. В следующих главах мы намерены описать мыслительный процесс, направляющий решения и показывающий практические шаги. Также будут приведены практические примеры.

## **Глава 5. Как выбрать стартовый поток создания ценности**

Выбор потока создания ценности для DevOps-преобразований заслуживает внимательного рассмотрения. Выбранный поток будет диктовать не только степень сложности преобразований, но также определит, кто будет вовлечен в процесс преобразования. И тем самым повлияет на то, как нам сформировать команды и как наилучшим образом распределить работников.

Другая задача сформулирована Майклом Рембеци, который помог провести DevOps-преобразования в 2009 г. Тогда он работал в компании Etsy директором по производству. Он заметил: «Мы должны подбирать проекты преобразования осторожно, потому что, находясь в процессе поиска и устранения неисправностей, мы ограничены в средствах. Поэтому нужно тщательно выбрать объекты, усовершенствование которых сильнее всего улучшит состояние всей организации».

Давайте проанализируем, как команда компании Nordstrom начала свои преобразования DevOps в 2013 г. Кортни Кисслер, вице-президент по электронной коммерции и технологиям хранения, описала этот опыт на конференциях DevOps Enterprise Summit, прошедших в 2014 и 2015 гг.

Основанная в 1901 г. Nordstrom была ведущим продавцом модной одежды, нацеленным на создание положительных впечатлений у своих клиентов от процесса покупки. В 2015 г. компания имела годовой доход в размере 13,5 миллиарда долларов.

Отправной точкой для DevOps-преобразований было, скорее всего, ежегодное заседание совета директоров в 2011 году. В том году одним из стратегических вопросов было обсуждение необходимости роста прибыли от онлайн-продаж. Совет изучил бедственное положение компаний Blockbusters, Borders и Barnes & Nobles, продемонстрировавших плачевые последствия того, что организации традиционной розничной торговли слишком поздно создают конкурентоспособные интернет-подразделения. Такие организации явно рисковали потерять позиции на рынке или даже полностью уйти из бизнеса.

В то время Кортни Кисслер была старшим директором по системам доставки и технологиям продаж, ответственным за значительную часть технологической организации работы, включая системы в обычных магазинах и веб-узле электронной коммерции. Как позже описывала Кисслер, «в 2011 г. технология организации работы Nordstrom была оптимизирована для экономии расходов, мы передали на аутсорсинг многие технологические функции, у нас был ежегодный цикл планирования с большими заданиями “каскадного” выпуска ПО. И хотя планы выполнялись на 97 % по показателям сроков выпуска, бюджета и достижения целей, мы были недостаточно оснащены для достижения целей пятилетней бизнес-стратегии, потребовавшейся от нас, после того как компания Nordstrom начала оптимизировать скорость работы, а не расходы».

Кисслер и группа технологии управления в Nordstrom были вынуждены принять решение, с чего именно начинать прикладывать усилия по трансформации. Они не хотели вызывать потрясений во всей системе. Вместо этого они хотели бы сосредоточить внимание на отдельных областях бизнеса, чтобы можно было экспериментировать и учиться. Группе была важна победа на первом этапе: она дала бы всем уверенность, что улучшения могут быть повторены в других областях бизнеса компании. Как именно этого достичь, было пока неизвестно.

Внимание сосредоточили на трех областях: на клиентском мобильном приложении, на системе ресторанов, расположенных в их магазинах, и на их цифровых данных. В каждой из этих сфер имелись бизнес-цели, недостижаемые при обычной организации работ. Легче было пробовать различные методы работы. И вот рассказ о первых двух.

Мобильное приложение Nordstrom при запуске потерпело неудачу. Как сказала Кисслер, «наши клиенты были крайне разочарованы продуктом, и когда мы запустили его в App Store, то получили много одинаковых отрицательных отзывов. Что еще хуже, существующие структуры и процессы (то есть “система”) были разработаны так, что обновления выпускались два раза в год». Другими словами, любые исправления попадали к клиенту спустя несколько месяцев.

Поэтому первой целью было делать выпуски быстрее или по требованию, обеспечивая более быстрые итерации и способность реагировать на обратную связь, предоставляемую клиентами. Разработчики создали специализированную продуктовую группу, предназначенную исключительно для поддержки мобильных приложений, при этом данная группа должна была самостоятельно реализовывать задачи, тестировать и доставлять ценность клиентам. Она больше не зависела от других и координировала свою деятельность с десятками групп внутри Nordstrom. Кроме того, был произведен переход от планирования на год к непрерывному процессу планирования. В результате появился единый список приоритетных работ по мобильному приложению на основе потребностей клиентов, что привело к исчезновению конфликта приоритетов, имевших место, когда группа поддерживала несколько продуктов.

На следующий год ликвидировали тестирование как отдельный этап работы: вместо этого его

интегрировали в повседневную деятельность. Вдвое увеличили количество функциональных возможностей, создаваемых за месяц, и вдвое уменьшили количество дефектов, добившись тем самым отличного результата.

Во второй области действий основное внимание было уделено системам поддержки расположенных в магазинах ресторанов под маркой Café Bistro. В отличие от потока создания ценности мобильного приложения, где необходимо было сократить время выхода на рынок и увеличить скорость разработки новых функциональных возможностей, здесь бизнес нуждался в уменьшении расходов и повышении качества. В 2013 г. компания Nordstrom завершила разработку 11 «концепций изменений в ресторанах», потребовавших изменений в работе соответствующих торговых точек, в результате чего увеличилось число жалоб клиентов. Вызывало тревогу, что на 2014 г. было запланировано внедрение 44 подобных концепций — в четыре раза больше, чем в предыдущем.

Как утверждала Кисслер, «один из руководителей бизнеса предложил команде устроить численность для работы с новыми требованиями, но я предложила не бросать дополнительных сотрудников на амбразуру, а вместо этого улучшить способы работы».

Команда смогла определить проблемные области — сбор информации и развертывание — и сосредоточила на них усилия по улучшению. Она смогла сократить время развертывания кода на 60 % и число ошибок, требующих вмешательства вручную, на 60–90 %.

Эти успехи дали группе уверенность: используемые принципы и методы DevOps могут быть применены к широкому кругу потоков создания ценности. Кисслер получила повышение — должность вице-президента по электронной коммерции и технологиям хранения в 2014 г.

В 2015 г. Кисслер заявила, что для достижения целей по продажам и развития технологий, ориентированных на клиентов, «необходимо увеличить продуктивность всех потоков создания ценности, а не только отдельных. На уровне управления мы сформировали общее по всей компании требование сокращения времени выполнения заказов на 20 % для всех услуг, ориентированных на клиентов».

«Это дерзкий вызов, — продолжила она. — У нас много проблем при текущем состоянии — процессы и времена циклов не совпадают у различных команд, они непрозрачны. Первое целевое состояние требует от нас помочь всем командам ввести одинаковую систему оценок, сделать процессы видимыми и проводить эксперименты, чтобы сокращать длительность процессов от итерации к итерации».

Кисслер пришла к следующему выводу: «Оценивая общую перспективу, мы считаем, что такие методы, как отображение потока создания ценности, уменьшение объема рабочих заданий до потока единичных работ, а также использование непрерывной поставки и микросервисов, приведут нас в нужное состояние. Однако в то же время мы все еще учимся, мы убеждены, что движемся в правильном направлении, и каждый знает, что усилия имеют поддержку на самых высоких уровнях руководства».

В этой главе представлены различные модели, дающие нам возможность воспроизвести мыслительные процессы, использованные командой компании Nordstrom, чтобы решить, с каких потоков создания ценности начинать. Мы будем оценивать потоки по многим критериям, в том числе являются ли они сервисами «чистыми» или «нечистыми», системами обязательств или системами записей. Мы будем также оценивать баланс между риском и вознаграждением для каждого преобразования и оценивать вероятный уровень сопротивления команд.

Мы часто категоризируем программные услуги или продукты как «чистые» (greenfield, «гринфилд») или как «нечистые» (brownfield, «браунфилд»). Эти обозначения первоначально использовались при планировании городов и строительных объектов. «Чистое» строительство ведется на неосвоенных землях. «Нечистым» называется такое, которое ведется на земле, ранее использовавшейся для промышленных целей, потенциально зараженной опасными отходами или загрязнениями. При застройке городов многие факторы могут сделать реализацию «чистых» проектов более простой, чем «нечистых», — нет конструкций, которые прежде необходимо демонтировать, и нет токсичных материалов, которые предварительно требуется вывезти.

В области технологий «чистый» проект — разработка нового программного продукта или инициативы, чаще всего на ранних этапах планирования или реализации. Мы создаем приложения и инфраструктуру с малым количеством ограничений. Начать проект разработки программного обеспечения с нуля проще, особенно если проект уже обеспечен финансированием и команда разработчиков либо создается, либо уже имеется. Кроме того, поскольку мы начинаем с нуля, то можем меньше беспокоиться о существующих кодовых базах, процессах и командах.

«Чистые» DevOps-проекты часто используются в качестве пилотных, демонстрируя осуществимость публичного или частного облака, экспериментальной автоматизации развертывания и аналогичных инструментов. Пример такого проекта — Hosted LabView, разработанный в 2009 г. в компании

National Instruments, существующей 30 лет, имеющей 5000 сотрудников и миллиард долларов годового дохода. Чтобы быстро вывести продукт на рынок, была создана новая команда разработчиков. Ей было разрешено не соблюдать существующие процессы и поручено изучить возможность использования публичных облаков. Первоначально в состав входили архитектор приложений, системный архитектор, два разработчика, разработчик системы автоматизации, руководитель команды и два специалиста из офшорного подразделения. Используя методы DevOps, они смогли вывести Hosted LabView на рынок за половину времени, обычно нужного для разработки продукта.

На другом конце спектра — «нечистые» DevOps-проекты. Это существующие продукты, уже находящиеся у клиентов, возможно, в течение многих лет или даже десятилетий. «Нечистые» проекты зачастую включают значительные объемы технического долга, например неавтоматизированное тестирование или работа на неподдерживаемых платформах. В примере с Nordstrom, приведенном выше, и система ресторанов в магазинах, и система электронной торговли были браунфилд-проектами.

Хотя многие считают, что DevOps предназначен в первую очередь для «чистых» проектов, он использовался и для успешного преобразования различного рода «нечистых» проектов. Фактически свыше 60 % проектов трансформации, о которых шла речь на конференции DevOps Enterprise Summit в 2014 г., были второго типа. В этих случаях существовал большой разрыв между потребностями клиентов и тем, что организации фактически давали. И DevOps-преобразования принесли огромные преимущества для бизнеса.

По сути, один из выводов доклада 2015 State of DevOps Report (о состоянии DevOps) подтвердил, что по возрасту приложения сложно предсказать его производительность. Для такого предсказания надо определить, было ли оно спроектировано (или перепроектировано) для обеспечения удобства тестирования и развертывания.

Команды, поддерживающие гринфилд-проекты, могут быть очень восприимчивы к экспериментированию с DevOps. В частности, в этой области широко распространено мнение, что традиционные методы недостаточны для достижения целей, и существует твердое ощущение неотложной необходимости совершенствования.

При преобразовании браунфилд-проектов можно столкнуться со значительными препятствиями и проблемами, особенно когда автоматическое тестирование не реализовано или когда используется сильно связанная архитектура, что не дает небольшим командам возможности выполнять разработку, тестирование и развертывание кода независимо друг от друга. Как мы можем преодолеть эти препятствия, обсуждается в книге.

Можно привести следующие примеры успешных преобразований браунфилд-проектов.

- **Компания CSG (2013 г.).** В 2013 г. компания CSG International получила 747 миллионов долларов дохода и имела более 3500 сотрудников, более 90 тысяч агентов службы поддержки клиентов для выставления счетов и обслуживания более чем 50 миллионов клиентов, пользующихся услугами видео- и голосовой связи и передачи данных. Каждый месяц выполнялось более шести миллиардов операций, печатались и рассыпались по почте более 70 миллионов бумажных счетов. Первоначально улучшения должны были охватить печать документов, одно из их главных занятий, использующее приложение для мейнфреймов, написанное на языке COBOL и связанное с двадцатью технологическими платформами. В качестве составной части трансформации компания приступила к выполнению ежедневных развертываний в среде, близкой к производственной, и в два раза чаще стала выпускать обновленные версии для клиентов, — не два, а четыре раза в год. В результате была значительно увеличена надежность приложения, а время развертывания кода сократилось с двух недель до менее чем одного дня.

- **Компания Etsy (2009 г.).** В 2009 г. компания Etsy имела 35 сотрудников и приносила доход в размере 87 миллионов долларов, но, после того как она едва выжила в сезон предрождественских распродаж, она начала преобразования практически по каждому направлению деятельности, в итоге превратившись в одну из наиболее уважаемых компаний, использующих DevOps, и создала условия для успешного размещения своих акций на бирже в 2015 г.

Исследовательская компания Gartner недавно популяризовала понятие «двуухровневые IT», ссылаясь на широкий спектр услуг, которые поддерживает типичное крупное предприятие. В рамках двухуровневого IT существуют *системы записей*, подобные ERP. Благодаря им функционирует наш бизнес (например, MRP, HR, системы финансовой отчетности), где корректность транзакций и данных имеет первостепенное значение, и *системы совместной работы*, включающие работу с клиентами и с сотрудниками (системы электронной торговли и приложения, обеспечивающие высокую производительность).

Системы записей обычно меняются медленнее и нередко должны соответствовать нормативным требованиям (например, закону Сарбейнза — Оксли). Компания Gartner называет такие системы «типа 1»; организации, их использующие, уделяют основное внимание установке «делать правильно».

Системы совместной деятельности обычно имеют гораздо более высокий темп изменений для поддержки быстрой обратной связи, с тем чтобы можно было проводить эксперименты и больше узнать о том, как удовлетворить потребности клиентов. Компания Gartner называет эти системы «типа 2»; организации, их использующие, уделяют основное внимание правилу «делать быстро».

Может оказаться удобным разделить системы на эти категории, однако мы знаем, что коренной, хронический конфликт между «делать правильно» и «делать быстро» можно разрешить, используя DevOps. Данные из докладов State of DevOps, которые готовит компания Puppet Labs, — следуя урокам бережливого производства — показывают: высокопроизводительные организации имеют возможность одновременно обеспечивать как производительность, так и надежность.

Более того, из-за взаимозависимости систем и способности вносить изменения в любую из них на обе накладываются ограничения, которые трудно изменить, а это почти всегда — система записей.

Скотт Праф, вице-президент отдела разработки компании CSG, отметил: «Мы приняли философию, отвергающую двухуровневое ИТ, поскольку все клиенты заслуживают и скорости, и качества. Это означает, что нам необходимо техническое совершенство, будь то группа поддержки 30-летнего приложения для мейнфрейма, приложения Java или мобильных приложений».

Поэтому, желая улучшить браунфилд-системы, мы должны не только стремиться уменьшить их сложность и улучшить надежность и стабильность, но и стараться сделать их быстрее, безопаснее и проще. Даже когда новые функциональные возможности добавлены только к «чистым» системам, они часто вызывают проблемы с надежностью в действующих системах записи, на которые они опираются. Делая эти низкоуровневые системы безопаснее для изменений, мы помогаем всей организации быстрее и безопаснее достичь целей.

В каждой организации обязательно найдутся команды и отдельные сотрудники с широким диапазоном стремлений и способностью воспринимать новые идеи. Джеки Мур первым обозначил этот диапазон как форму жизненного цикла внедрения технологий в книге «Преодоление пропасти. Маркетинг и продажа хайтек-товаров массовому потребителю», где «пропасть» — классическая невозможность добраться до тех, кто не относится к новаторам или первопроходцам (рис. 9).

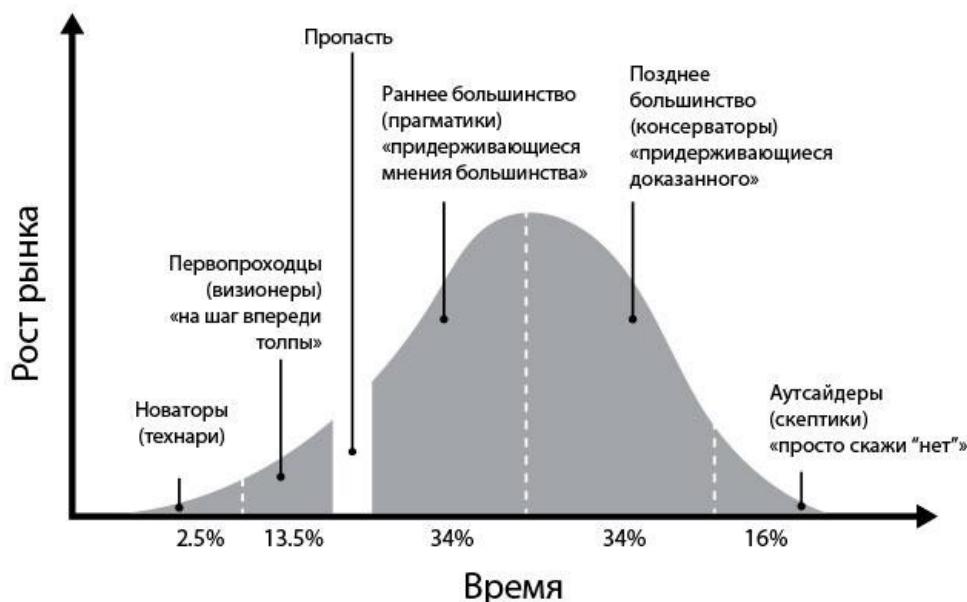


Рис. 9. Кривая внедрения технологий (источник: Мур и Маккена. Преодоление пропасти. Маркетинг и продажа хайтек-товаров массовому потребителю, Crossing The Chasm, 15)

Другими словами, новые идеи часто быстро берут на вооружение новаторы и первопроходцы, а носители более консервативных взглядов противостоят им (раннее большинство, позднее большинство и аутсайдеры). Наша цель — найти группы, убежденные в необходимости применять принципы и методы DevOps, обладающие желанием сделать новое и продемонстрировавшие

способность к новаторству и совершенствованию собственных процессов. В идеале эти группы будут с энтузиазмом поддерживать путешествие в страну DevOps.

Не будем тратить время на попытки преобразований в более консервативных группах, особенно на начальных этапах. Вместо этого направим энергию на создание успеха у групп с меньшими рисками и сформируем базу, опираясь на них (этот процесс будет рассмотрен в следующем разделе). Даже если имеется поддержка со стороны высокого руководства, будем избегать революционного подхода — «большого шока» (то есть новации сразу и повсюду). Вместо этого сосредоточим усилия в нескольких областях деятельности организации, обеспечив тем самым успех инициатив и возможность расширения успеха.

Независимо от того, как мы оцениваем сферу наших первоначальных усилий, мы на самых ранних этапах должны продемонстрировать достижения и транслировать их на всю организацию. Мы делаем это, разбивая крупные цели по улучшению на небольшие последовательные шаги. Это не только дает возможность быстрее выполнять усовершенствования, но также позволяет узнать, когда мы сделали неправильный выбор потока создания ценности. Обнаружив ошибки на ранних этапах, мы можем быстро вернуться назад и повторить попытку, пробуя разные новые решения с учетом сделанных выводов.

Генерируя успехи, мы получаем право расширить масштабы инициативы DevOps. Мы хотим продвигаться по безопасной последовательности, повышающей уровень доверия к нам, степень нашего влияния и получаемой поддержки. Приведенный ниже список позаимствован из курса лекций доктора Роберто Фернандеса, адаптированного Уильямом Паундсом, профессором менеджмента Массачусетского технологического института. Здесь описаны идеальные фазы, применяемые, чтобы побуждать ответственных лиц создавать и расширять коалиции и обеспечивать базу поддержки преобразований.

**1. Найдите новаторов и первопроходцев.** Поначалу мы сосредоточиваем усилия на командах, действительно желающих помочь, — это родственные души и друзья по путешествию, и они первыми добровольно захотят пойти в страну DevOps. В идеале это специалисты, пользующиеся уважением и имеющие большое влияние на остальную часть организации, что повышает доверие к нашей инициативе.

**2. Создайте авторитетную массу и молчаливое большинство.** На следующем этапе мы стремимся расширить методы DevOps на большее число команд и потоков создания ценности с целью формирования стабильной базы поддержки. Работая с группами, чутко реагирующими на наши идеи, — даже если эти группы не самые заметные или влиятельные, — мы расширяем коалицию, а она увеличивает успех, создавая эффект присоединения к большинству, что еще больше повышает наше влияние. Мы специально обходим опасные политические баталии, ставящие под угрозу нашу инициативу.

**3. Выявите уклоняющихся.** «Уклоняющиеся» — высококлассные и авторитетные специалисты, но они будут, скорее всего, противостоять нашим усилиям (возможно, даже саботировать их). В целом мы обращаемся к этой группе только после того, как привлекли на свою сторону молчаливое большинство, когда в достаточной степени достигли успеха и можем хорошо выступить в защиту нашей инициативы.

Расширение DevOps на всю организацию — непростая задача. Оно может вызвать нежелательные последствия для отдельных сотрудников, отделов и даже для организации в целом. Но, как говорит Рон Ван Кеменад, директор по информационным технологиям компании ING, преобразовавший свою компанию в одну из наиболее популярных организаций в области технологий, «лидерство в изменениях требует мужества, особенно в корпоративных средах, где сотрудники боятся изменений и будут противостоять им. Но если вы начнете с малого, то вам действительно нечего бояться. Любой лидер должен быть достаточно смел, чтобы выделить команды на решение задач, связанных с риском, хотя и предусмотренным».

Питер Друкер, лидер в области разработки методов обучения менеджменту, отмечал: «Маленькая рыбка учится быть крупной в маленьких прудах». Тщательно выбирая, где и как начать, мы смогли определить в организации участки, создающие продукт и не затрагивающие при этом деятельность остальных частей организации. Поступая таким образом, мы создаем себе базу поддержки, зарабатываем право расширить использование DevOps и получаем признание и благодарность заинтересованных лиц (сотрудников, клиентов и т. п.).

## **Глава 6. Основные сведения о работе в потоке создания ценности, превращении его в прозрачный и расширении на всю организацию**

Определив, в каком потоке создания ценности мы хотим применить принципы и модели DevOps, мы должны получить достаточное представление, как продукт будет доставляться заказчику: какое задание выполняется и кем, какие шаги можно предпринять для оптимизации потока.

В предыдущей главе мы рассказали о проведенных компанией Nordstrom преобразованиях DevOps под руководством Кортни Кисслер и ее группы. В процессе многолетней работы они выяснили: один из наиболее эффективных способов начать улучшение любого потока создания ценности — провести рабочее совещание с основными заинтересованными сторонами и выполнить упражнения по отображению потока создания ценности. Этот процесс описан в данной главе. Он поможет зафиксировать все шаги создания ценности.

Отображение потока создания ценности может привести к ценным и неожиданным открытиям. Кисслер любит пример, когда команда пыталась уменьшить длительность времени обработки заказов, выполнявшихся через систему «офиса косметики». Написанное на языке COBOL приложение для мейнфреймов обеспечивало деятельность менеджеров в отделах красоты и косметики в магазинах компании.

Приложение давало возможность менеджерам регистрировать новых продавцов для линий продуктов, продававшихся в магазинах компании, чтобы можно было отслеживать комиссии от продаж, давать скидки, предоставленные вендором, и т. д.

Кисслер объясняла:

«Я очень хорошо знала это приложение, раньше я занималась поддержкой этой технологической команды. Почти десять лет в ходе каждого ежегодного цикла планирования по программам мы обсуждали, как перенести это приложение с мейнфрейма на другую платформу. Разумеется, как и в большинстве организаций, даже при наличии полной поддержки руководства мы никогда не могли выкроить время.

Моя команда хотела провести отображение потока создания ценности, чтобы определить, действительно ли приложение на COBOL создает проблему, или, возможно, существует более масштабная проблема, и ее нам необходимо выявить. Провели совещание. Собрали всех, кто имел хотя бы какое-то отношение к доставке продукта внутренним клиентам и партнерам по бизнесу, — членов группы обслуживания мейнфрейма, группы общей поддержки и т. д.

Обнаружили, что, когда менеджеры отдела отправляли форму запроса “назначение исполнителя на линию продукции”, а мы запрашивали его табельный номер, им неизвестный, они либо оставляли соответствующее поле пустым, либо вводили в него текст наподобие “я не знаю”. Что еще хуже, для заполнения формы менеджерам надо было идти из складского помещения в офис, где стояли компьютеры. В результате получалась напрасная трата времени, а запрос несколько раз перебрасывался туда-сюда».

В ходе совещания участники провели несколько экспериментов, в том числе исключили поле с номером сотрудника в форме и позволили другому отделу получать информацию с более низкого уровня. Эксперименты, проведенные при помощи менеджеров соответствующего отдела, показали: время обработки запроса сократилось на четыре дня. Группа позднее заменила приложение для ПК приложением для iPad, что дало возможность менеджерам передавать необходимую информацию, не покидая складское помещение, а время обработки было позднее уменьшено до секунд.

Кисслер с гордостью заявляет: «После потрясающих улучшений требования перенести приложение с мейнфрейма прекратились. Кроме того, руководители других направлений отметили этот факт и стали приходить к нам с длинными списками дальнейших экспериментов, которые они хотели бы провести при нашем участии в своих организациях. Каждый член деловых и технологических команд восхищался результатами, поскольку решались реальные проблемы бизнеса, а самое главное, в процессе решения исполнители узнавали нечто новое для себя».

Далее мы опишем следующие этапы: определение команд, необходимых для создания потока ценности для клиентов, формирование карты потока создания ценности, на которой видны все требуемые операции, и использование карты как руководства для команд, демонстрирующего, как лучше и быстрее создавать продукт. Так мы сможем повторить потрясающие результаты, описанные на примере компании Nordstrom.

Пример компании Nordstrom показывает: как бы ни был сложен поток создания ценности, ни один человек не знает всего списка задач, решаемых в ходе создания ценности для клиента, особенно если необходимая работа выполняется различными командами, нередко разделенными организационным устройством компании, географическим фактором или методами стимулирования.

В результате после выбора приложения или сервиса для преобразований DevOps мы должны определить всех участников потока создания ценности, несущих ответственность за совместно создаваемую ценность для клиентов. В общем случае в этот список входят:

- **владелец продукта.** Представитель компании, определяющий набор функциональности в следующей версии продукта;
- **разработчики.** Команда, ответственная за разработку функциональности продукта;
- **служба качества.** Команда, ответственная за работу петь обратной связи и обеспечивающая, чтобы продукт функционировал так, как задумано;
- **эксплуатация.** Команда, отвечающая за поддержание производственной среды в рабочем состоянии и помогающая обеспечить требуемый уровень обслуживания;
- **отдел информационной безопасности.** Команда, отвечающая за обеспечение безопасности систем и данных;
- **менеджеры релиза.** Специалисты, ответственные за управление и координацию развертывания в производство и процессы релиза;
- **технические руководители или менеджеры потока создания ценности.** В литературе, посвященной бережливому производству, так называют тех, кто «обеспечивает соответствие потока создания ценности требованиям клиента на всех этапах процесса».

Определив членов потока создания ценности, сделаем следующий шаг — получим конкретное понимание, как выполняется то, что задокументировано в виде карты потока создания ценности. В потоке создания ценности деятельность начинается с владельца продукта — он либо предоставляет запрос от клиента, либо предлагает новую бизнес-гипотезу. Некоторое время спустя она будет передана разработчикам, они воплотят функции в код и внесут код в хранилище контроля версий. Затем сборки кода будут интегрированы, протестированы в среде, близкой к производственной, и, наконец, развернуты в производство, где они (в идеале) начнут создавать продукт для клиентов.

Во многих традиционных организациях поток создания ценности будет состоять из тысяч шагов, требующих участия сотен людей. Поскольку документирование любой карты потока создания ценности такой сложности потребует, скорее всего, не одного дня, можно проводить совещание несколько дней, пока наконец не установим всех вовлеченных лиц и не перестанем отвлекать их от повседневной деятельности.

Наша цель не документировать каждый шаг до мельчайших подробностей, а понять, какие именно области потока создания ценности ставят под угрозу наши цели — быстрый ход потока, короткое время выполнения заказов и получение результатов клиентами. В идеальном случае мы должны собрать исполнителей, обладающих полномочиями изменить поток создания ценности.

Деймон Эдвардс, соведущий подкаста DevOps Cafe, отмечал: «По моему опыту, упражнения по составлению карт потоков создания ценности всегда открывают людям глаза. Нередко они при этом впервые обращают внимание, как много надо сделать и насколько чрезмерные усилия приложить, чтобы доставить продукт клиенту. Может оказаться, что отдел эксплуатации в первый раз видит последствия того, что разработчики не имеют доступа к правильно настроенной среде, и это создает еще более сумасшедшие условия во время развертывания кода. Отдел разработки может впервые увидеть, какие чудовищные усилия приходится прикладывать тестировщикам и отделу эксплуатации для развертывания их кода в производстве: им приходится еще долго заниматься им после того, как некоторая функция помечена флагком “выполнено”».

Воспользовавшись всем спектром знаний каждой из команд, вовлеченных в поток создания ценности, мы должны сосредоточить пристальное внимание на следующих областях исследования.

- Места, где работа может простаивать в очереди недели или даже месяцы, такие как получение среды, близкой к производственной, изменение процесса одобрения или процессы проверки безопасности.
- Места, где производятся повторная работа или переделка либо возникает потребность в них.

На первом этапе документирования потока создания ценности должны рассматриваться только блоки высокого уровня. Обычно даже для сложных потоков создания ценности группы могут

сформировать схему с числом блоков от 5 до 15 в течение нескольких часов. Каждый блок процесса должен включать в себя время выполнения заказа и время производства элемента, над которым ведется работа, а также показатель %C/A, измеренный на уровне конечных клиентов.

Мы используем количественные показатели из карты потока создания ценности, чтобы верно направить усилия по улучшению потока. В компании Nordstrom, например, внимание было сосредоточено на низком показателе %C/A в форме запроса, отправляемого менеджерами без указания табельного номера. В других случаях это может быть длительное время выполнения заказа или низкое значение %C/A при предоставлении командам разработчиков правильно настроенных тестовых сред. Или это может быть связано с длительным периодом выполнения регрессионного тестирования перед релизом новой версии программного обеспечения.

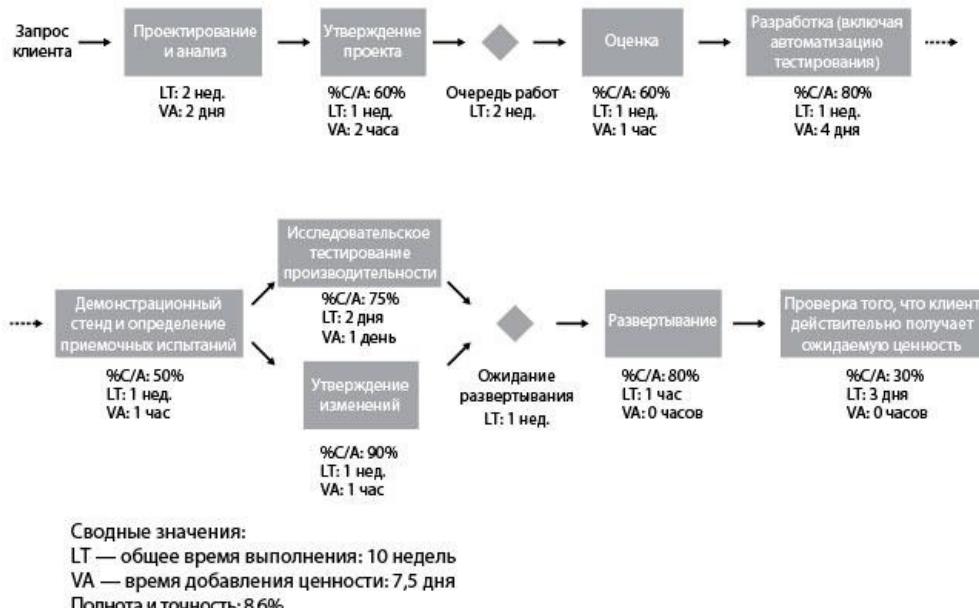


Рис. 10. Пример карты потока создания ценности (источник: Humble, Molesky, and O'Reilly, Lean Enterprise, 139)

Определив показатели для улучшения, надо перейти на следующий уровень наблюдений и измерений для лучшего понимания проблемы. Затем строить идеализированную будущую карту потока создания ценностей. Она послужит описанием целевого состояния к определенной дате (обычно в срок от трех до двенадцати месяцев).

Руководители помогают определить будущее состояние и затем направляют команду и дают ей возможность выполнить мозговой штурм для выдвижения гипотез и разработки контрмер в целях достижения желаемых улучшений для движения к этому состоянию, провести эксперименты для проверки гипотез и обработать результаты, чтобы понять, правильны ли он. Команда повторяет эти действия, используя сделанные выводы при проведении следующих экспериментов.

Одна из серьезных проблем инициатив, подобных DevOps, — то, что они неизбежно входят в конфликт с текущей деловой активностью. Отчасти это естественный результат успешного развития бизнеса. Организация, успешная в течение любого продолжительного периода (годы, десятилетия или даже столетия), создает механизмы для увековечения методов, способствовавших ее благополучию, таких как разработка продукции, управление заказами и работа с каналами поставок.

Для закрепления и защиты механизмов существующих процессов используется много разных методов (специализация, сосредоточенность на эффективности и воспроизводимости, бюрократия, требующая одобрения всех процессов, и контроль для защиты от изменений). В частности, бюрократия невероятно устойчива и предназначена для выживания в неблагоприятных условиях — можно уволить половину чиновников, а процесс будет идти по-прежнему.

Хотя это хорошо для сохранения статус-кво, часто требуется изменить методы, чтобы приспособиться к изменяющимся условиям на рынке. Что, в свою очередь, требует инноваций и разрушения старых приемов. Возникает противоречие с интересами групп, в настоящее время отвечающих за повседневные операции и внутренние бюрократические механизмы и почти всегда выходящих победителями из внутренних конфликтов.

В своей книге *The Other Side of Innovation: Solving the Execution Challenge*, Виджей Говиндараджан и Крис Тримбл, преподаватели Такской школы бизнеса в Дартмутском колледже, описали тематические исследования по вопросу, как вводить инновации без разрушений, несмотря на мощную инерцию повседневной деятельности. Они документально фиксировали, как автомобильные страховки по методу «продвижение клиента» были успешно разработаны и продавались компанией Allstate, как прибыльный цифровой издательский бизнес был создан газетой *Wall Street Journal*, как модифицировались невиданные кроссовки для бега по пересеченной местности, разработанные компанией *Timberland*, и как развивался первый электромобиль компании *BMW*.

Основываясь на своих исследованиях, Говиндараджан и Тримбл утверждают: организациям необходимо создать специальную команду преобразований, способную работать без связи с остальными частями организации, несущими ответственность (они называют эти части «выделенная группа» и «двигатель производительности» соответственно).

Прежде всего мы будем возлагать на специализированную группу ответственность за достижение четко определенных и поддающихся измерению результатов на уровне системы (например, уменьшить время развертывания от «код зафиксирован в системе контроля версий» до «код успешно работает в производственной среде» на 50 %). Чтобы реализовать такую инициативу, выполним следующие действия.

- Назначим членов выделенной команды, чья деятельность будет направлена исключительно на усилия по трансформации DevOps (в отличие от распространенной практики вроде «все ваши текущие обязанности сохраняются, но 20 % вашего времени вы должны тратить на эту новую штучку DevOps»).
- Выберем среди членов команды специалистов широкого профиля, имеющих навыки в различных областях.
- Выберем членов команды, имеющих давние и взаимные связи с другими отделами организации.
- Выделим, если это возможно, специализированной команде отдельное помещение для максимального увеличения обмена информацией внутри команды и создания некоторой изоляции от остальных частей организации.

Если возможно, освободим команду преобразования от целого ряда правил и процедур, ограничивающих действия других отделов организации, как это сделали в компании *National Instruments*, описанной в предыдущей главе. В конце концов, сложившиеся в компании процессы — одна из форм коллективной памяти, и мы нуждаемся в специальной команде для создания новых процессов и знаний, необходимых для получения желаемых результатов и создания новой коллективной памяти.

Создание специализированной команды полезно не только для нее самой, но также и для «двигателя производительности». Формируя отдельную команду, мы создаем пространство для экспериментов с новыми методиками, одновременно защищая остальные части организации от возможных сбоев и отвлекающих факторов.

Одна из наиболее важных составляющих любой инициативы по улучшению — определение измеримой цели с четко заданным сроком достижения (от шести месяцев до двух лет). Она должна требовать значительных усилий, но быть все же реалистичной. И достижение цели должно создать очевидную ценность как для организации в целом, так и для клиентов.

Эти цели и сроки должны быть согласованы с руководителями и стать известны всем в организации. Мы также хотим ограничить число типов инициатив, реализуемых одновременно, чтобы они не стали чрезмерно обременительными для управленческих возможностей руководителей, занимающихся организационными изменениями, и для самой компании. В качестве примеров целей для улучшения можно привести следующие.

- Снизить процентную долю расходов на поддержку продуктов и незапланированные работы на 50 %.
- Обеспечить срок с момента фиксации кода до релиза его в производство не более недели для 95 % изменений.
- Обеспечить релиз кода в производство только в обычное рабочее время с нулевым временем

простоя.

- Интегрировать все необходимые управляющие элементы контроля безопасности в конвейер развертывания, чтобы обеспечить проверку соответствия всем необходимым требованиям.

Когда все четко уяснят цель высокого уровня, командам следует принять решение о регулярном темпе работы по улучшению. Мы хотим, чтобы преобразования осуществлялись подобно тому, как ведется разработка продукта, то есть итеративно, с постепенным наращиванием результата. Длительность типичной итерации будет находиться в интервале от двух до четырех недель. При каждой итерации команды должны договориться о небольшом наборе целей, генерирующих ценность и создающих некоторый прогресс на пути к долгосрочной цели. В конце каждой итерации командам следует проанализировать достигнутый прогресс и установить новые цели для следующей итерации.

В любом проекте преобразования DevOps мы должны сохранить горизонты планирования близкими, как если бы речь шла о стартапе, выпускающем продукт или выполняющем разработку для клиента. Инициативы должны быть нацелены на создание измеримых улучшений или получение рабочих данных в течение нескольких недель (в крайнем случае — месяцев).

Продолжая планировать на небольшие интервалы времени и поддерживая короткие итерации, мы достигаем следующего:

- гибкости и способности быстро изменить приоритеты и планы;
- уменьшения интервала между временем, когда были затрачены усилия, и реализацией улучшений, что укрепляет петлю обратной связи и повышает вероятность того, что она упрочит желаемое поведение — когда инициативы по улучшению успешны, это поощряет дополнительные инвестиции;
- более быстрого обучения, начинающегося с первой итерации, что означает более быструю интеграцию выводов на следующем шаге;
- уменьшения усилий, необходимых для создания улучшений;
- более быстрой реализации усовершенствований, значительно влияющих на повседневную деятельность;
- сокращение риска того, что проект погибнет, прежде чем мы сможем получить какие-либо ощутимые результаты.

Общая проблема — правильное определение приоритетов. В конце концов, организации, больше всего в этом нуждающиеся, имеют меньше всего времени на улучшение. Это особенно верно в технологических организациях из-за проблемы технического долга.

Организации, борющиеся с финансовыми долгами только путем внесения процентных платежей и никогда не сокращающие величину основного долга, могут в итоге оказаться в ситуации, когда уже будут не в силах обслуживать даже процентные платежи. Точно так же организации, не уменьшающие технический долг, могут обнаружить: они настолько перегружены ежедневным поиском обходных путей решения проблем, что уже не способны завершить никакую новую работу. Другими словами, в такой ситуации они только платят проценты по техническому долгу.

Мы будем активно управлять техническим долгом, расходуя по меньшей мере 20 % всех усилий в области разработки и управления эксплуатацией на рефакторинг, средства автоматизации, улучшение архитектуры и нефункциональные требования (NFR), такие как сопровождаемость, управляемость, масштабируемость и надежность, пригодность к тестированию и развертыванию, а также безопасность.

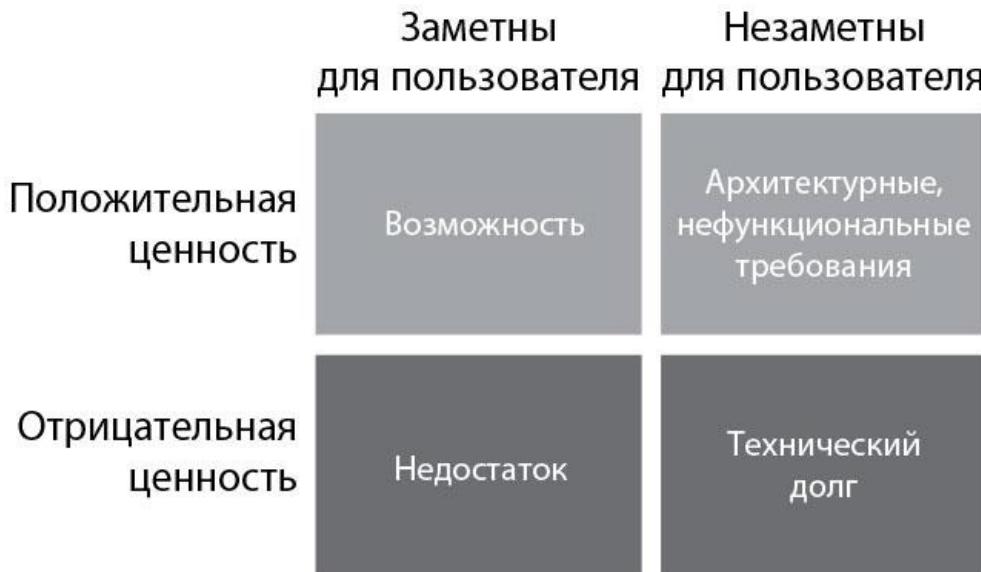


Рис. 11. Вкладывайте 20 % времени в создание положительных ценностей, даже если они незаметны для пользователя (источник: запись Machine Learning and Technical Debt with D. Sculley в подкасте Software Engineering Daily, November 17, 2015, )

После изучения опыта компании eBay, оказавшейся в конце 1990-х гг. практически на грани исчезновения, Марти Кэган, автор книги *Inspired: How To Create Products Customers Love*, считающейся наиболее фундаментальным исследованием по конструированию продукции и управлению, пришел к такому выводу.

«Сделка с владельцами и инженерами заключается обычно так: отдел управления производством отнимает у руководителей верхнего звена право распоряжаться примерно 20 % времени инженерной группы и дает ей право тратить его так, как она посчитает нужным. Это время можно использовать, чтобы переписать код заново, перепроектировать систему или выполнить рефакторинг проблемных частей кода... все, что они считают необходимым сделать во избежание появления нужды прийти к команде и признаться: "Мы должны остановиться и переписать весь код". Если ситуация действительно тяжелая, то на эти цели можно отдать 30 % ресурсов или даже больше. Однако я начинаю нервничать, когда сталкиваюсь с командами, уверенными, будто справятся с подобными задачами, получая гораздо меньше ресурсов, чем 20 %».

Кэган отмечает, что если организации не выплачивают «двадцатипроцентный сбор», то размер технического долга увеличится до такой степени, что неизбежно придется тратить все время на его погашение. В какой-то момент оказываемые услуги становятся настолько легкоразрушаемыми, что добавление новых функциональных возможностей останавливается, поскольку все инженеры стремятся обеспечить надежность или обойти возникающие проблемы.

Выделяя 20 % времени на то, чтобы разработка и отдел эксплуатации могли создавать эффективные контрмеры против проблем, с которыми мы сталкиваемся в повседневной деятельности, мы можем обеспечить ситуацию, когда технический долг не будет препятствовать нашей способности быстро и безопасно разрабатывать и предоставлять услуги. Накопление проблем усиливает давление технического долга на исполнителей и даже может вызвать у них нервное истощение.

Операция InVersion компании LinkedIn демонстрирует интересное тематическое исследование, свидетельствующее, что необходимость выплачивать технический долг — часть повседневной деятельности. Спустя шесть месяцев после успешного проведения IPO в 2011 г. LinkedIn продолжала бороться с проблемами развертывания: оно стало настолько болезненным, что они запустили операцию InVersion, и разработка новых функциональных возможностей и развитие имеющихся были полностью остановлены на два месяца. Все силы были брошены на пересмотр конфигурации вычислительных сред, процедур развертывания и архитектуры.

Компания LinkedIn была создана в 2003 г., чтобы пользователи могли подключаться к сети для улучшения возможностей поиска работы. К концу первой недели существования в ней насчитывалось 2700 участников. Спустя год их число превысило миллион и с тех пор росло экспоненциально. В ноябре 2015 г. в LinkedIn было зарегистрировано свыше 350 миллионов человек, создающих десятки тысяч запросов в секунду, в результате чего на серверы данных LinkedIn каждую секунду поступают миллионы запросов.

Вначале LinkedIn использовала в основном доморощенное приложение Leo, монолитное

приложение Java, обслуживавшее каждую страницу с помощью сервлетов и управляемых JDBC соединений с различными внутренними базами данных Oracle. Однако для удовлетворения растущего трафика в первые годы работы компании две важнейшие услуги были отделены от Leo: первая обрабатывала запросы о соединениях участника и делала это полностью в памяти, а вторая — поиск участников — опиралась на первую.

К 2010 г. большинство новых разработок сопровождалось созданием новых служб, и уже почти сто таких служб функционировало за пределами Leo. Проблема заключалась в том, что обновления в Leo развертывались только раз в две недели.

Джон Клемм, старший технический менеджер LinkedIn, пояснил, что к 2010 г. в компании накопилось значительное количество проблем с Leo. Несмотря на вертикальное масштабирование этого приложения путем добавления памяти и процессоров, «Leo часто давал сбои, было трудно найти причину неполадки и восстановить работу, сложно добавить новый код... Нам было ясно, что необходимо “убить Leo” и разделить на много небольших функциональных и не влияющих друг на друга служб».

В 2013 г. журналист Эшли Вэнс из агентства Bloomberg описывал: «Если бы LinkedIn попытался добавить группу новых функций одновременно, сайт рухнул бы и превратился в груду обломков, и инженерам пришлось бы работать ночами, устранивая возникшие проблемы». К концу 2011 г. работа до глубокой ночи была уже не чем-то из ряда вон выходящим, поскольку проблемы приобрели грандиозный масштаб. Некоторые из инженеров верхнего уровня компании, включая Кевина Скотта, пришедшего в LinkedIn на должность технического директора за три месяца до того, как сайт компании начал свою деятельность, решили полностью остановить работу над новыми функциями и перевести весь отдел разработки на укрепление основной инфраструктуры сайта. Они назвали это операцией InVersion.

Скотт начал операцию InVersion как путь к «внедрению зачатка культурного манифеста в инженерную культуру его команды. Никакая новая функция не будет разрабатываться, пока компьютерная архитектура LinkedIn не будет переделана — именно это нужно для бизнеса компании и ее команды».

Скотт так описывал одну из неприятных сторон этого решения: «Вы начали всем видимую деятельность, весь мир смотрит на вас, и тут мы заявляем руководству, что не собираемся делать ничего нового, так как все инженеры будут работать над проектом [InVersion] два следующих месяца. Это пугает».

Однако Вэнс рассказал о значительном положительном результате операции InVersion. «LinkedIn создал целый пакет программного обеспечения и инструментов, помогающих разрабатывать код для его сайта. Вместо того чтобы ждать несколько недель, пока новые функции проделают свой путь на главный сайт LinkedIn, инженеры могли разработать новый сервис, использовать ряд автоматизированных систем изучения кода с целью поиска ошибок в коде и проблем при взаимодействии сервиса с существующими функциями и запустить его прямо на сайте LinkedIn... Инженерная служба LinkedIn теперь выполняет крупные обновления сайта трижды в день». За счет создания более безопасной системы создаваемая продукция обеспечивает меньший объем сверхурочной работы по ночам и большее количество времени для разработки новых, инновационных функций.

Как писал Джош Клемм в статье о масштабировании в LinkedIn, оно «может быть измерено через многочисленные аспекты, в том числе организационные... Операция InVersion позволила всей инженерной организации сосредоточить усилия на улучшении и инструментов, и разворачивания, и инфраструктуры, и производительности труда разработчиков. Она была успешной в деле предоставления инженерам гибкости, в которой мы нуждались для создания масштабируемых новых продуктов, имеющихся у нас сегодня... В 2010 г. мы уже имели более 150 отдельных сервисов. Сегодня у нас есть более 750 сервисов».

Кевин Скотт заявил: «Ваша задача как инженера и ваши цели как технологической команды — помочь компании выиграть. Если вы руководите группой инженеров, лучше, чтобы вы нацеливались на перспективы, намечаемые CEO. Ваша задача — выяснить, в чем же нуждается компания, бизнес, рынок, конкурентоспособная производственная среда. Примените это знание к вашей инженерной группе, чтобы ваша компания смогла выиграть».

Дав компании LinkedIn возможность выплатить накапливавшийся в течение почти десятилетия технический долг, проект InVersion обеспечил стабильность и безопасность следующего этапа роста компании. Вместе с тем он потребовал двух месяцев общей сфокусированности на нефункциональных требованиях в ущерб всем ранее обещанным в ходе IPO функциональным возможностям. Сделав поиск и устранение проблем частью повседневности, мы управляем техническим долгом, чтобы избежать ситуаций, когда вдруг окажемся «на грани исчезновения».

Чтобы иметь возможность знать, добиваемся ли мы прогресса в достижении цели, важно, чтобы

каждый сотрудник организации видел текущее состояние работы. Существует множество способов сделать текущее состояние прозрачным, но самое важное — актуальность показанной информации и постоянный анализ: какие показатели мы измеряем, чтобы убедиться, что они помогают нам понять прогресс в достижении условий, понятых как цель.

В следующем разделе обсуждаются модели, способные помочь обеспечить наглядность работы и синхронизировать ее между различными командами и функциями.

Как отметил Кристофер Литл, технический администратор и один из самых первых летописцев DevOps, «антропологи описывают инструменты как культурные артефакты. Любое обсуждение культуры после приручения огня должно также сопровождаться обсуждением инструментов». Точно так же в потоке создания ценности DevOps мы используем инструменты для укрепления культуры и ускорения желаемых изменений поведения.

Одна из целей — сделать так, чтобы инструменты подкрепляли не только общие цели разработчиков и отдела эксплуатации, но и создавали общий бэклог задач, вписанный в общую рабочую систему и использующий общую терминологию. Следовательно, задачи могут быть расставлены по приоритетам на глобальном уровне.

Действуя таким образом, разработчики и ИТ-отдел могут в конечном счете создать общую очередьность работ, вместо того чтобы каждый использовал свою систему (например, разработчики используют JIRA, а отдел эксплуатации — ServiceNow). Значительное преимущество такого решения в том, что инциденты на производстве регистрируются в той же системе, что и текущая работа, и очевидно, что при наличии в системе инцидентов следует прекратить другую деятельность, особенно если мы используем доску канбан.

Другое преимущество использования общих инструментов разработчиками и сотрудниками отдела эксплуатации — общая очередьность работ, где каждый определяет приоритеты проектов улучшений с глобальной точки зрения, выбирая наиболее значимые для организации или максимально сокращающие технический долг. Обнаружив технический долг, мы стараемся погасить его немедленно или же добавляем его в приоритетную очередь. В отношении проблем, оставшихся неучтеными, мы можем использовать «20 % времени на нефункциональные требования» для исправления проблем из верхней части очереди.

Другие технологии, укрепляющие общие цели, — чаты, например IRC, HipChat, Campfire, Slack, Flowdock и OpenFire. Чат позволяет быстро обмениваться информацией (в отличие от заполнения форм, обрабатывающих с помощью предварительно определенных рабочих процессов), дает возможность пригласить других специалистов по мере необходимости и вести журналы истории, заполняющиеся автоматически. Они могут быть проанализированы при разборе событий.

Когда мы имеем механизм, позволяющий любому члену команды быстро помочь другим участникам группы, даже специалистам из других команд, создается удивительный динамический эффект: время, необходимое для получения информации, или знания, что именно должно быть сделано, можно сократить с нескольких дней до нескольких минут. Кроме того, поскольку все записывается, нам может не потребоваться в будущем просить кого-то о помощи — мы сами сумеем отыскать нужную информацию.

Однако среда быстрых коммуникаций, предоставляемая чатом, может иметь и обратную сторону. Как отмечает Райан Мартенс, основатель и технический директор компании Rally Software, «в чате, если кто-либо не получает ответа через пару минут, считается нормой, что можно отправить вопрос снова и делать это, пока вы не получите необходимый ответ».

Надежда на немедленное реагирование может, конечно, привести к нежелательным результатам. Постоянный шквал запросов и вопросов, мешающий работе, может не дать инженерам вовремя выполнить запланированное. В результате группы могут принять решение, что определенные типы запросов должны отправляться через более структурированные и асинхронные инструменты.

В этой главе мы определили команды, поддерживающие поток создания ценности, отразили на карте потока создания ценности то, что требуется, чтобы доставить продукт клиенту. Карта создает основу понимания текущего состояния, включая время выполнения заказа и параметры %C/A для проблемных областей, и информирует, как достичь будущего состояния.

Это позволяет группам, выделенным для преобразований, быстро выполнять итерации и проводить эксперименты для повышения производительности. Мы также убедились, что сможем выделить достаточное количество времени для улучшения, исправления известных проблем и ограничений архитектуры, в том числе и на нефункциональные требования. Примеры с компаниями Nordstrom и LinkedIn продемонстрировали, как значительно можно уменьшить время выполнения заказов и улучшить качество, если мы найдем проблемы в потоке создания ценности и выплатим технический долг.

## Глава 7. Как проектировать организацию и ее архитектуру, не забывая о законе Конвея

В предыдущих главах мы определили поток создания ценности для запуска программы преобразований DevOps и установили общие цели и методы для подключения отобранный для преобразований команды в целях улучшения способов доставить продукт клиенту.

В этой главе мы начнем разбираться, как лучше организовать себя для достижения целей потока создания ценностей. В конце концов то, как мы организуем команду, повлияет на то, как мы будем работать. В 1968 году доктор Мелвин Конвей провел известный эксперимент в контрактной исследовательской организации. Восьми сотрудникам было поручено написать компиляторы языков COBOL и ALGOL. Он отметил: «После некоторых первоначальных оценок сложности и необходимого времени пять человек были назначены на написание компилятора COBOL, и три — компилятора ALGOL. В результате компилятор COBOL выполнял компиляцию в пять проходов, компилятор ALGOL — в три».

Эти наблюдения послужили основой того, что сейчас называют законом Конвея, гласящего: «Организации, проектирующие системы... производят их, копируя структуры коммуникаций, сложившиеся в этих организациях... Чем крупнее организация, тем менее она гибкая и тем сильнее выражен характер этого явления». Эрик Реймонд, автор книги *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, придумал упрощенный (и сейчас более известный) вариант закона Конвея, изложенный в словаре Jargon File: «Организация программного обеспечения и организация команды разработчиков программного обеспечения будут совпадать»; обычно упрощается до «если над компилятором работают четыре группы, то получите четырехпроходный компилятор».

Другими словами, то, как мы организуем команды, оказывает мощное воздействие на производимое программное обеспечение, равно как и на результаты проектирования и производства. Чтобы процесс от разработчиков к отделу эксплуатации протекал быстро, с высоким качеством и большой отдачей для клиентов, мы должны организовать команды и деятельность так, чтобы закон Конвея работал на нас. Если сделали плохо, то закон не даст командам действовать безопасно и самостоятельно: вместо этого они будут тесно связаны друг с другом и ожидать друг от друга завершения назначенных частей общей задачи, и даже небольшие изменения потенциально вызовут глобальные, катастрофические последствия.

Пример того, как закон Конвея может либо препятствовать достижению целей, либо способствовать этому, можно найти в технологии Sprouter, разработанной компанией Etsy. Etsy начала использовать DevOps в 2009 г. Сейчас она одна из наиболее уважаемых организаций с DevOps, в 2014 г. получила доход в размере почти 200 миллионов долларов и успешно провела IPO в 2015 г.

Первоначально разработанная в 2007 г., Sprouter соединяла людей, процессы и технологии. Способы соединения нередко приносили нежелательные результаты. Sprouter, сокращение английской фразы stored procedure router (маршрутизатор хранимых процедур), была первоначально разработана для облегчения жизни разработчиков и групп, поддерживающих базы данных. Как заявил во время своей презентации на конференции Surge 2011 Росс Снайдер, старший инженер компании Etsy, «Sprouter была разработана, чтобы команды разработчиков могли писать код PHP для приложений, а администраторы баз данных — запросы SQL в Postgres и Sprouter помогала бы им встретиться на середине этих процессов».

Sprouter размещался между фронтэнд-приложениями PHP и базой данных Postgres, обеспечивая централизованный доступ к базе данных и скрывая реализацию БД от приложений прикладного уровня. Проблема в том, что внесение любых изменений в бизнес-логику приводило к значительным трениям между разработчиками и администраторами баз данных. Как заметил Снайдер, «почти для любой новой функциональной возможности сайта требовалось, чтобы администраторы баз данных написали для Sprouter новую хранимую процедуру. В результате каждый раз, когда разработчики хотели добавить новые функциональные возможности, им было необходимо содействие администраторов баз данных, а его нередко нельзя получить, минуя множество бюрократических процедур». Другими словами, существует зависимость разработчиков, создающих новые функциональные возможности, от администраторов баз данных. Задачи, порожденные этой зависимостью, должны в списке приоритетных быть скоординированы с разработчиками и осуществляться во взаимодействии с ними. В результате работа ждет своей очереди, время тратится на совещаниях, а итог появляется намного позже, чем мог бы. Это происходит из-за того, что Sprouter создал тесную взаимосвязь между разработчиками и администраторами баз данных, не давая разработчикам возможность самостоятельно разрабатывать, тестировать и развертывать их код в производство.

Кроме того, процедуры, хранящиеся в базе данных, были тесно связаны с Sprouter: при любом изменении хранимой процедуры необходимо было также внести изменения в сам Sprouter. В результате этого Sprouter еще чаще становился центром возникновения сбоев. Снайдер объяснил, что все было так тесно связано и требовало такого высокого уровня синхронизации, что в

результате почти каждое развертывание приводило к мини-простою.

Обе проблемы связаны со Sprouter, и их возможное решение может быть объяснено законом Конвея. В компании Etsy первоначально было две команды, разработчики и администраторы баз данных. Каждая отвечала за два уровня служб: уровень логики приложений и уровень хранимых процедур. Две группы работали над двумя уровнями именно так, как предсказывает закон Конвея. Sprouter был предназначен для облегчения жизни обеим командам, но он работал не так, как ожидалось: стоило бизнес-правилам измениться, как вместо изменений на двух уровнях потребовались изменения в трех (приложение, процедуры, а теперь еще и в Sprouter). В результате сложность координации и определения приоритетности между тремя командами значительно возросла и вызвала проблемы с надежностью.

Весной 2009 г. в ходе того, что Снайдер называл «великим культурным преобразованием в компании Etsy», к компании в качестве технического директора присоединился Чад Дикерсон. Дикерсон привел в движение множество вещей, включая крупные инвестиции в стабильность сайта, дал разработчикам возможность самостоятельно выполнять развертывание кода в производство, а также начал занявшую два года ликвидацию Sprouter.

Для этого команда решила переместить всю обработку бизнес-логики из уровня базы данных на уровень приложений, устранив необходимость Sprouter. Исполнители создали небольшую группу, написавшую на языке PHP уровень объектно-реляционного сопоставления (ORM — Object Relational Mapping), что позволило разработчикам внешнего интерфейса обращаться непосредственно к базе данных и сократило число команд, задействованных в изменении бизнес-логики, с трех до одной.

Как описывал Снайдер, «мы начали использовать ORM для любых новых областей сайта и постепенно переносили небольшие части нашего сайта из Sprouter в ORM. Нам потребовалось два года для переноса всего сайта и отключения Sprouter. И хотя все это время мы недовольно ворчали на Sprouter, он, несмотря на это, оставался в производственной среде».

За счет устранения Sprouter были также ликвидированы проблемы, связанные с ситуациями, когда несколько команд нуждаются в координации для внесения изменений в бизнес-логику. Сократилось количество случаев передачи заданий от одной команды к другой, значительно увеличилась скорость и успешность внедрения в производство, повысилась стабильность сайта. Кроме того, поскольку небольшие команды теперь могут самостоятельно разрабатывать и развертывать код, не привлекая новые команды для внесения изменений в другие области системы, увеличилась производительность труда разработчиков.

Sprouter был удален из производственной среды и репозиториев версионного контроля компании Etsy в 2001 г. Как сказал Снайдер, «класс! Отличные ощущения!».

Как показал опыт Снайдера и компании Etsy, то, как мы организовали работу в нашей компании, определяет, каким образом будет выполняться задача и, следовательно, каких результатов мы сможем достичь. В оставшейся части главы мы будем изучать вопрос, как закон Конвея может отрицательно влиять на производительность потока создания ценности и, что более важно, как организовать команды, чтобы обернуть закон Конвея себе на пользу.

В теории принятия решений существуют три основных типа организационных структур. Это подсказывает нам, как разрабатывать потоки создания ценности DevOps, держа при этом в голове закон Конвея. Три типа структур называются «функциональная», «матричная» и «рыночная». Роберто Фернандес определяет их так.

- Функциональные организации оптимизируются во имя компетентности, разделения труда или сокращения затрат. Эти компании централизуют знания, помогающие служебному росту и развитию навыков, и нередко имеют длинную иерархическую организационную структуру. Долгое время это был преобладающий метод организации для эксплуатации (например, администраторов серверов, сетевых администраторов, администраторов баз данных и так далее — все они были собраны в отдельные группы).
- Матрично-ориентированные организации пытаются объединить функциональную и рыночную ориентации. Однако, как могли наблюдать многие, работавшие в матрично-ориентированных организациях или руководившие ими, такие попытки часто приводят к образованию сложных организационных структур, таких как подчинение исполнителя сразу двум или большему числу менеджеров, и случается, что компании не удается достичь целей ни по одному из двух направлений.
- Организации, ориентированные на рынок, нацелены быстро реагировать на потребности клиентов. Как правило, структура таких организаций плоская, состоящая из нескольких кросс-функциональных направлений (например, маркетинга, проектирования и т. д.), что зачастую

приводит к потенциальной избыточности во всей организации. Именно такой метод применяется во многих известных компаниях, внедривших принципы DevOps. Яркие примеры — Amazon и Netflix: каждая команда одновременно отвечает и за реализацию функций доставки, и за поддержку.

Постоянно имея в виду эти три категории организаций, давайте изучим, как чрезмерная функциональная ориентированность, особенно в эксплуатации, может привести к нежелательным результатам в технологическом потоке создания ценности, как мог бы предсказать закон Конвея.

Традиционные организации, занимающиеся IT-эксплуатацией, создавая команды с учетом специализации, часто используют функциональную ориентацию. Мы можем объединить администраторов баз данных в одной группе, сетевых администраторов в другой, администраторов серверов в третьей и т. д. Одним из наиболее заметных последствий таких действий будет длительное время выполнения задач, особенно в случае сложных мероприятий, таких как большие развертывания, где мы должны будем создать заявки для нескольких групп и координировать передачу заданий между ними, в результате чего на каждом шаге времени, затраченное на данную работу, будет увеличиваться за счет длинных очередей.

Проблема обычно усугубляется тем, что исполнитель часто не видит задачу целиком или не понимает, как его деятельность связана с целями потока создания ценности (например, «я конфигурирую серверы, просто потому что мне это поручили»). Это помешает работникам в творческий и инновационный вакуум.

Проблема обостряется, если каждый функциональный отдел эксплуатации обслуживает несколько потоков создания ценности (то есть несколько команд разработчиков), конкурирующих между собой за ограниченные ресурсы. Чтобы команды разработчиков могли выполнять задачи по графику, нередко приходится привлекать к решению проблем менеджера или директора, а порой и руководителя более высокого уровня (обычно исполнительного директора). Он сможет наконец определить приоритеты согласно глобальным целям организации, а не функциональным задачам подразделений. Это решение должно затем пройти по цепочке в каждой из функциональных областей для изменения местных приоритетов, а это замедлит деятельность других групп. И хотя каждая команда ускоряет работу, в результате все проекты будут двигаться к завершению черепашьим шагом.

В дополнение к очередям и длительному выполнению такая ситуация приводит к медленной передаче заданий из отдела в отдел, большому количеству требуемых переделок, проблемам с качеством, возникновению узких мест и задержек.

Тупиковая ситуация препятствует достижению важных целей организации, что зачастую значительно перевешивает стремление к сокращению расходов.

Также функциональную ориентацию можно обнаружить в централизации контроля качества и функций информационной безопасности, работающих хорошо (или по крайней мере достаточно неплохо) и при не очень частых релизах программного обеспечения. Однако по мере того как мы будем увеличивать количество команд разработчиков и частоту выполняемых развертываний и релизов, организации из числа наиболее функционально-ориентированных столкнутся с затруднениями при поддержании удовлетворительных результатов, особенно когда все выполняется вручную. А теперь изучим, как действуют организации, ориентированные на рынок.

Говоря в широком смысле, для достижения результатов DevOps нам требуется уменьшить последствия функциональной ориентации («оптимизация затрат») и перейти к рыночной ориентации («оптимизация скорости»), с тем чтобы мы могли иметь много небольших команд, работающих безопасно и независимо друг от друга, быстро доставляя продукт клиентам.

Суть в том, что команды, ориентированные на рынок, несут ответственность не только за развитие функций, но и за тестирование, безопасность, развертывание и поддержку сервисов в производстве, от рождения идеи до прекращения ее использования. Эти группы созданы кросс-функциональными и независимыми, они имеют возможность разрабатывать и запускать пользовательские эксперименты, разрабатывать и поставлять новые функции, развертывать и запускать службы в производственной среде, устранять любые дефекты без зависимости от других групп, давая им таким образом возможность продвигаться к цели быстрее. Такая модель принята компаниями Amazon и Netflix и до сих пор провозглашается первой из них как одна из основных причин способности организации быстро меняться даже в процессе роста.

Для достижения рыночной ориентации не требуется проводить серьезную реорганизацию компании сверху донизу, часто создающую большое количество сбоев, вызывающую страх и даже парализующую деятельность организации. Вместо этого мы будем внедрять функциональных инженеров с соответствующими навыками (например, эксплуатация, тестирование, информационная безопасность) в каждую сервисную команду или предоставлять эти возможности

командам с помощью автоматизированных самообслуживающихся платформ, воспроизводящих приближенную к производственной среде, выполняющих автоматизированное тестирование или производящих развертывание.

Это позволяет каждой сервисной команде независимо доставлять продукт клиентам без необходимости создавать задачи для других групп, например IT-эксплуатации, тестирования или информационной безопасности.

Если вы располагаете только рекомендованными ориентированными на рынок командами, то можете создать эффективные и действующие с высокой скоростью организации, имеющие функциональную ориентацию. Кросс-функциональные и ориентированные на рынок команды — один из способов, но не единственный, создать быстрый и надежный поток. Мы можем также достичь желаемых результатов DevOps, используя функциональную ориентацию, пока каждый в потоке создания ценности не начнет рассматривать результаты работы клиентов и самой организации в качестве общей цели, независимо от того, какую роль он исполняет в компании.

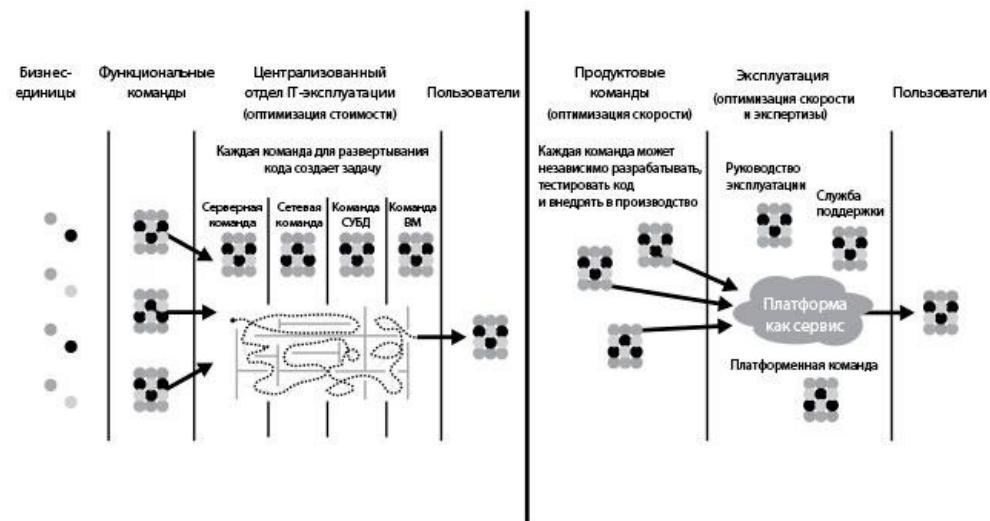


Рис. 12. Сравнение функциональной и рыночной ориентаций.

Слева: функциональная ориентация — все рабочие потоки проходят через централизованный отдел ИТ-эксплуатации; справа: рыночная ориентация — все продуктовые команды могут самостоятельно развертывать в производство свои компоненты, слабо связанные с другими (источник: Humble, Molesky, and O'Reilly, Lean Enterprise, Kindle edition, 4523 & 4592).

Например, высокая производительность с функционально ориентированной и централизованной ИТ-группой возможна, если сервисные команды гарантированно и быстро получают то, что им необходимо (в идеале — по первому требованию), и наоборот. Многие из наиболее известных организаций, использующих DevOps, в том числе компании Etsy, Google и GitHub, сохраняют функциональную ориентацию отдела эксплуатации.

У этих организаций есть общая черта — культура высокого доверия, позволяющая всем отделам эффективно действовать вместе. Все видят, как распределяются приоритеты. Существует резерв ресурсов для обеспечения быстрого завершения высокоприоритетных задач. Это возможно в том числе благодаря автоматизированным самообслуживающимся платформам, обеспечивающим качество всех разрабатываемых продуктов.

При анализе «бережливого производства» (Lean Manufacturing), зародившегося в 1980-е гг., многие исследователи были озадачены функциональной ориентацией компании Toyota, не совпадавшей с практикой организации кросс-функциональных, ориентированных на рынок групп. Они были удивлены этим, как они говорили, «вторым парадоксом компании Toyota».

Как писал Майк Ротер в уже упоминавшейся книге «Тойота Ката. Лидерство, менеджмент и развитие сотрудников для достижения выдающихся результатов», «как бы соблазнительно это никазалось, нельзя реорганизовать путь к постоянному совершенствованию и приспособляемости. Решающий фактор — не форма организации, а то, как люди действуют и реагируют. Корни успеха компании Toyota лежат не в организационной структуре, но в развивающихся возможностях и особенностях ее работников. Это удивляет многих, обнаруживающих, что компания Toyota в основном организована традиционно — как система функциональных отделов». Развитие привычек и возможностей у отдельных инженеров и у персонала в целом будет рассматриваться в следующих

разделах.

В высокопроизводительных организациях каждый член команды разделяет общую цель — качество, доступность и безопасность не область ответственности конкретных отделов, но часть каждого ежедневного задания.

Это означает, что неотложной проблемой текущего дня может быть разработка или развертывание клиентской функции или устранение инцидента на производстве, имеющего первостепенную важность. Или же может потребоваться дать оценку решению, принятому коллегой-инженером, применить срочное исправление системы безопасности на производственных серверах либо сделать усовершенствование, позволяющее коллегам действовать более продуктивно.

С учетом общих целей и разработчиков, и отдела эксплуатации технический директор компании Ticketmaster Джоди Малки заявил: «Почти 25 лет я использовал метафоры из американского футбола для описания разработчиков (Dev) и эксплуатации (Ops). Вы знаете, что Ops — это оборона, не дающая пропускать голы, а Dev — нападение, старающееся голы забивать. И в один прекрасный день я понял, как ошибочна эта метафора: они никогда не играют одновременно. Фактически они даже не одна команда!»

Он продолжает: «Теперь я использую другие аналогии. Ops — игроки нападения, а Dev — квотербеки и принимающие игроки, чья деятельность заключается в перемещении мяча по полю. Задание Ops — обеспечить Dev достаточно времени для надлежащего проведения игры».

Яркий пример того, как общая боль может усилить стремление к достижению общих целей, — взрывной рост компаний Facebook в 2009 г. Она испытывала серьезные проблемы с развертыванием кода, и хотя не все они вызывали проблемы у клиентов, работа походила на постоянное многочасовое тушение пожаров. Педро Канаухати, директор по организации производства в Facebook, описывал совещание с участием множества инженеров отдела эксплуатации: кто-то попросил тех, кто не трудится сейчас над исправлением инцидента, закрыть свои ноутбуки, но никто этого не сделал.

Одной из наиболее важных вещей, сделанных с целью помочь изменить результаты развертывания, была организация дежурств всех инженеров в Facebook, технических руководителей и разработчиков архитектуры для помощи в устраниении неполадок в сервисах. При этом каждый работавший над сервисом получал очень жесткую и нелицеприятную обратную связь о принятых им архитектурных решениях и написанном коде, что оказало положительное воздействие на результаты.

Что касается организаций с функционально ориентированными подразделениями, то это отделы специалистов (администраторы сети, администраторы систем хранения данных и т. д.). Когда такие отделы узкоспециализированы, это вызывает их самоизоляцию, которую Спир описывает так: «работают подобно суверенным государствам». Любая сложная деятельность при этом требует неоднократной передачи задач от отдела к отделу, что ведет к очередям и, следовательно, к более длительному выполнению (например, потому что любое изменение конфигурации сети должно быть сделано сотрудником сетевого отдела).

Поскольку мы зависим от постоянно растущего числа технологий, постольку должны иметь в штате инженеров, специализирующихся в нужных нам областях технологии и достигших в них мастерства. Однако при этом мы не хотим воспитывать специалистов, застрявших в прошлом, знающих только одну область потока создания ценности и способных внести свой вклад только в нее.

Одна из контрмер — поощрять стремление каждого члена команды стать универсалом. Мы делаем это, предоставляя инженерам возможность приобрести навыки, необходимые для создания систем, за которые они отвечают, путем регулярной ротации сотрудников по разным ролям. Термин «специалист по комплексной разработке» (full stack engineer) в настоящее время часто используется (иногда как богатая тема для пародий) для описания специалиста широкого профиля, знакомого со всем набором используемых приложений (например, языком, на котором написаны коды приложений, баз данных, операционных систем, сетевых решений, облаков) или по крайней мере имеющего понимание всего этого на базовом уровне.

Таблица 2. Сравнение узких специалистов с универсалами и с «Е-образными» сотрудниками (опыт, знания, исследования и выполнение)

<b>«I-образный» (специалист)</b>	<b>«T-образный» (универсал)</b>	<b>«Е-образный»</b>
Глубокие знания и опыт в одной области	Глубокие знания и опыт в одной области	Глубокие знания в нескольких областях
Очень мало навыков и опыта в других областях	Широкий набор навыков в разных областях	Опыт во многих областях. Признанное мастерство. Всегда занимается инновациями
Быстро создает узкие места	Может сделать улучшение для устранения узкого места	Потенциал практически безграничен
Не обращает внимания на простой, возникающие на следующей стадии, и свой вклад в них	Обращает внимание на простой, возникающие на следующей стадии, и свой вклад в них	
Мешает гибкому планированию или сглаживанию изменчивости процесса	Помогает сделать планирование гибким и сглаживает изменчивость процесса	

**Источник:** Scott Prugh, страница Continuous Delivery на сайте , 14 февраля 2013 г., .

Скотт Праф писал, что компания CSG International претерпела трансформацию, обеспечившую нужные ресурсы для создания и запуска продукта в рамках одной команды, включая анализ, проектирование, разработку, тестирование и внедрение в производство. «Благодаря кросс-профессиональной подготовке и улучшающимся инженерным навыкам универсалы могут выполнять заказы с большим объемом, чем их коллеги-специалисты, а также увеличивают общий поток задач благодаря устраниению очередей и времени ожидания». Этот подход идет вразрез с традиционной практикой найма, но, как объясняет Праф, оно того стоит. «Традиционные руководители будут часто выражать против найма на работу инженеров с универсальным набором навыков, утверждая, что они обходятся дороже и что гораздо разумнее нанять двух администраторов серверов вместо одного инженера-универсала. Однако преимущества, которые дает бизнесу возможность ускорения протекания потока деятельности, перевешивают». Кроме того, как отмечает Праф, «вложения в кросс-профессиональное обучение правильны для карьерного роста сотрудников и делают работу каждого более приятной».

Когда мы ценим людей лишь за их навыки или производительность в нынешней роли, а не за способность приобретать и применять новые навыки, мы (часто непреднамеренно) усиливаем то, что Кэрол Двек описывает как *фиксированное сознание*: сотрудники рассматривают свой интеллект и способности как статическую «данность», которая не может быть изменена.

Вместо этого мы хотим поощрять обучение, помогать преодолеть беспокойство, помочь обрасти соответствующие навыки, составить план карьерного роста и т. д. Тем самым мы содействуем *росту осознанности* у инженеров — в конце концов, обучающейся организации нужны обучающиеся работники. Поощряя каждого, а также проводя обучение и оказывая в этом помощь, мы формируем наиболее устойчивый и наименее дорогостоящий способ создать ощущение собственной силы у команд, вкладываясь в обучение уже имеющихся работников.

Как писал Джейсон Кокс, директор по разработке систем в компании Disney, «в отделе

эксплуатации мы должны были изменить методы найма. Мы искали людей, обладавших “любопытством, мужеством и откровенностью”, которые могут быть не только универсалами, но также теми, кто отвергает общепринятые взгляды... мы хотим поощрять положительные потрясения, с тем чтобы наш бизнес не топтался на месте, а мог двигаться в будущее». Как мы увидим в следующем разделе, то, как мы вкладываемся в команды, тоже влияет на результаты.

Другой способ обеспечить высокопроизводительные результаты — создать стабильные сервисные команды с постоянным финансированием для выполнения их собственных стратегий и планов. В таких группах действуют специально выделенные инженеры, необходимые для релиза продукции согласно конкретным обязательствам перед внутренними и внешними заказчиками, такими как функциональные возможности, требования, составленные клиентами, и задачи.

Сопоставьте этот подход с более традиционным, где команды разработчиков и тестировщиков назначаются на проект, а затем перебрасываются на другой сразу после завершения первого проекта и прекращения его финансирования. Это приводит к различного рода нежелательным результатам, включая то, что разработчики не могут увидеть долгосрочные последствия принятых решений (форма обратной связи), и такую модель финансирования, где учитываются и оплачиваются расходы лишь на ранних стадиях жизненного цикла программного обеспечения, а он, к сожалению, стоит меньше всего в случае успешного продукта или сервиса.

При использовании модели финансирования «по продукту» цель — обеспечить достижение организационных результатов, а также результатов, получаемых и клиентом: это доход, цена обслуживания, темп восприятия продукта клиентом в идеале с минимальными затратами (например, количество усилий или времени, строк кода). Сравните это с тем, как обычно оцениваются проекты, например, был ли он завершен в рамках выделенного бюджета, в установленные сроки или в заданном объеме.

По мере роста организаций одной из крупнейших задач становится поддержание эффективной коммуникации и координации между людьми и командами. Когда люди и команды находятся на разных этажах, в различных зданиях или даже часовых поясах, создать и поддержать общее понимание и взаимное доверие сложно, и это препятствие для эффективной совместной деятельности. Сотрудничеству также мешают ситуации, когда основные механизмы коммуникации — задания на работу и запросы на изменения. Еще хуже, когда команды разделены установленными в договоре границами, как в случае выполнения заданий аутсорсинговыми командами.

Как мы видели на примере Sproutler в компании Etsy, описанном в начале этой главы, способ формирования команд может давать плохие результаты — таков побочный эффект закона Конвея. В число таких способов входит разделение групп по функциям (например, размещение разработчиков и тестировщиков в разных местах или полная передача тестирования на аутсорсинг) либо по слоям архитектуры (например, приложения, базы данных).

Такие конфигурации требуют значительных коммуникаций и координации между группами, но по-прежнему приводят к большим объемам переделок, разногласиям в спецификациях, неудовлетворительной передаче работы из одних отделов в другие и к тому, что специалисты пристаивают, ожидая завершения смежниками их части проекта.

В идеале архитектура программного обеспечения должна позволить небольшим командам действовать независимо, продуктивно и быть достаточно отделенными друг от друга, так чтобы работа могла быть сделана без чрезмерных или вовсе ненужных коммуникаций и координации.

Когда мы имеем сильно связанную архитектуру, небольшие изменения могут привести к крупномасштабным сбоям. В результате все, кто имеет дело с какой-то одной частью системы, должны постоянно координировать с другими отделами все задачи, влияющие на другие части системы, и эта координация осуществляется через сложные бюрократические процедуры управления изменениями.

Кроме того, проверка слаженности системы требует учета изменений, внесенных сотнями или даже тысячами других разработчиков, а они могут, в свою очередь, зависеть от десятков, сотен и тысяч взаимосвязанных систем. Тестирование осуществляется в условиях ограниченной интеграции тестовых сред, часто требующих недель на настройку. В результате не только требуется много времени на внедрение изменений (обычно недели или месяцы), но также низкими оказываются производительность труда разработчиков и результаты развертывания.

Напротив, если архитектура позволяет небольшим командам разработчиков реализовывать, тестировать и развертывать код в производство независимо, быстро и безопасно, мы можем увеличить и поддерживать на высоком уровне продуктивность разработчиков и улучшить результаты развертывания. Такими характеристиками обладает *сервис-ориентированная архитектура* (SOA), впервые описанная в 1990-е гг., в которой сервисы тестируются и развертываются независимо друг от друга. Ключевая особенность архитектуры SOA заключается в

том, что она состоит из слабосвязанных сервисов с ограниченными контекстами .

Слабосвязанная архитектура означает, что сервисы можно обновлять в производстве независимо друг от друга, без необходимости обновления других сервисов. Сервисы должны быть отделены от других сервисов и, что не менее важно, от общих баз данных (хотя они могут совместно использовать сервисы баз данных при условии, что они не имеют никаких общих схем).

Ограниченные контексты описаны в книге Эрика Эванса «Предметно-ориентированное проектирование (DDD)». Структуризация сложных программных систем». Идея в том, что разработчики должны быть в состоянии понять и обновить код сервиса, не зная ничего об устройстве других сервисов, вместе с которыми он функционирует. Сервисы взаимодействуют друг с другом только через API и не имеют общих структур данных, схем баз данных или других внутренних представлений объектов. Ограниченные контексты обеспечивают разграничения между сервисами и имеют хорошо определенные интерфейсы, что, ко всему прочему, упрощает тестирование.

Рэнди Шуп, бывший технический директор Google App Engine, отметил: «Организации с этими типами сервис-ориентированной архитектуры, такие как Google и Amazon, невероятно гибки и масштабируемы. Эти организации имеют десятки тысяч разработчиков, небольшие группы которых все еще способны быть невероятно продуктивными».

Закон Конвея помогает провести границы между командами в контексте требуемой модели коммуникаций, но он также призывает сохранять малые размеры команд, сокращая объем коммуникаций внутри и поощряя поддерживать область работы каждой команды ограниченной и четко очерченной.

В рамках своей инициативы по преобразованию и уходу от монолитной базы кода, начатой в 2002 г., компания Amazon использовала для определения размеров команд правило «двух пицц» (2PT — two-pizza team) — команда должна иметь такой размер, чтобы всех ее участников можно было накормить двумя пиццами (это обычно от пяти до десяти человек).

Такое ограничение размера порождает четыре следствия.

1. Оно обеспечивает наличие у команды четкого общего понимания системы, над которой она работает. Если команда становится больше, то количество коммуникаций, необходимых для того, чтобы каждый знал, что происходит, растет комбинаторно.

2. Оно ограничивает разрастание продукта или сервиса, над которым ведется работа. Ограничивая размер команды, мы ограничиваем скорость, с которой их система может развиваться. Это также помогает обеспечить общее понимание системы членами команды.

3. Оно децентрализует производительность и делает возможной автономную деятельность. Каждая команда «двух пицц» функционирует настолько автономно, насколько возможно. Руководитель команды, действуя совместно с менеджерами компании, решает, за какие ключевые бизнес-метрики, или функции пригодности, отвечает команда, и эти функции становятся общим критерием оценки тех экспериментов, которые проводит команда. После этого группа сможет действовать автономно, чтобы максимизировать эти показатели.

4. Руководство командой 2PT — способ для сотрудников получить некоторый опыт работы на руководящих должностях в среде, где сбои не имеют катастрофических последствий. Важным элементом стратегии Amazon была связь между организационной структурой 2PT и архитектурным подходом, взятым от сервис-ориентированной архитектуры.

Технический директор компании Amazon Вернер Фогельс в 2005 г. объяснил преимущества такой структуры Ларри Дигнану, сотруднику сайта Baseline. Дигнан пишет:

«Небольшие команды действуют быстро и не увязают в так называемом администрировании... Каждая команда назначена на отдельный участок бизнеса и полностью за него отвечает... Она оценивает объем задач по исправлению ошибки, разрабатывает исправление, внедряет его и контролирует его повседневное использование. Таким образом, программисты и архитекторы получают прямую обратную связь от потребителей кода или приложений — в ходе регулярных совещаний и неофициальных бесед».

Еще один пример того, как архитектура может коренным образом улучшить производительность, — это программа API Enablement компании Target.

Target — шестая по величине компания розничной торговли в США. Ежегодно она тратит на

технологии более миллиарда долларов. Хэзер Микман, директор развития компании, так описывала начало использования DevOps: «В недобрые старые дни бывало так, что работа серверов поддерживалась десятью различными командами, и когда что-то ломалось, мы, как правило, приостанавливали ее, чтобы не возникло дальнейших проблем, что, конечно, еще более ухудшало ситуацию».

Затруднения были вызваны тем, что получение сред и выполнение развертывания создали значительные сложности для разработчиков, такие как получение доступа к необходимым базам данных. Как описывала Микман, «проблема заключалась в том, что большая часть наших основных данных, таких как информация о материальных запасах, ценообразовании и магазинах, была “заперта” в устаревших системах и мейнфреймах. Нередко у нас оказывалось несколько источников одних и тех же данных, особенно при взаимодействии системы электронной торговли и физических хранилищ, которые принадлежали различным командам с различными структурами данных и имевшими различные приоритеты... В результате если новая группа разработчиков хотела создать что-то для наших пользователей, то требовалось от трех до шести месяцев только для того, чтобы осуществить интеграцию с работающими системами для получения необходимых данных. Мало того, еще от трех до шести месяцев требовалось для тестирования вручную, чтобы убедиться, что они не сломали ничего критически важного, поскольку система была очень сильно связанной и имела множество пользовательских соединений типа точка — точка. Управление взаимодействием 20-30 различных команд с учетом существующих зависимостей требовало большого числа руководителей проектов для организации координации и передачи задач между командами. Это означает, что разработчики тратили все время на ожидание, когда придет их очередь, вместо того чтобы обеспечивать результаты.

Длительное время разработки при необходимости извлечения и создания данных в системах хранения ставило под угрозу важные цели бизнеса, такие как организация цепочек снабжения розничных магазинов компании Target и ее сайта электронных продаж. Для этого было необходимо получить каталог имеющихся товаров и работниками магазина, и клиентами, делающими заказы через сайт. Это загружало цепочку поставок намного сильнее пределов, предусмотренных при ее разработке, — первоначальной задачей было управлять движением товаром между поставщиками, распределительными складами и магазинами.

Пытаясь решить проблему с данными, Микман создала в 2012 г. команду API Enablement, чтобы команды разработчиков могли «обеспечивать новые возможности в течение нескольких дней, а не месяцев». Они хотели, чтобы каждая инженерная команда внутри компании Target имела возможность получать и хранить необходимые им данные, такие как информация о продукции или о магазинах, в том числе о часах работы, местоположении, о том, имеется ли в магазине кафе Starbucks и т. д.

Нехватка времени играла большую роль при выборе членов команды. Микман объяснила это так:

«Поскольку наша команда должна была добавлять новые возможности в течение дней, а не месяцев, мне нужна была команда, которая могла бы сделать работу сама, а не передавать ее подрядчикам — мы хотели набрать людей с выдающимися инженерными навыками, а не тех, кто знал, как управлять контрактами. И чтобы работа не простаивала в очередях, нам необходимо быть владельцами всего стека, что означает, что мы взяли на себя еще и работу по эксплуатации... Мы придумали много новых инструментов для поддержки непрерывной интеграции и непрерывной доставки. И поскольку мы знали, что если добьемся успеха, то у нас появится возможность расширения, причем чрезвычайно высокого, мы стали использовать и такие новые инструменты, как база данных Cassandra и система обмена сообщениями Kafka. Когда мы обратились за разрешением, нам отказали, но мы все равно сделали это, поскольку знали, что нам это нужно».

За следующие два года команда API Enablement добавила 53 новые возможности для бизнеса, в том числе Ship to Store и Gift Registry, а также их интеграцию с Instacart и Pinterest. Как описывала Микман, «неожиданно взаимодействовать с командой Pinterest стало очень легко, потому что мы просто предоставили им API».

В 2014 г. команда API Enablement обслуживала более полутора миллиардов вызовов API в месяц. К 2015 г. это возросло до 17 миллиардов вызовов в месяц, объединяя 90 различных API. Для поддержки этой возможности команда регулярно выполняла 80 развертываний в неделю.

Благодаря изменениям были созданы основные преимущества для бизнеса компании Target — рост цифровых продаж на 42 % в течение праздничного сезона 2014 г. и увеличение еще на 32 % во втором квартале следующего года. В 2015 г. в «черную пятницу» и в последовавшие за ней выходные было оформлено свыше 280 000 заказов с получением их в магазине. К 2015 г. целью компании было увеличить число магазинов, способных выполнять интернет-заказы, со 100 до 450 при общем количестве магазинов 1800.

«Команда API Enablement показывает, что может сделать команда инициативных специалистов, — говорит Микман. — И это помогает настроить нас на следующий этап, который заключается в

расширении использования DevOps на все технологии организации».

Из примеров компаний Etsy и Target мы видим, как значительно архитектура и проектирование организационной структуры могут улучшить результаты. Неправильно примененный закон Конвея приведет к тому, что компания получит плохие результаты, не обретя безопасность и гибкость. При правильном применении организация даст возможность разработчикам независимо и бесстрашно разрабатывать, тестировать и развертывать продукт для клиента.

## **Глава 8. Как получить лучшие результаты, интегрируя эксплуатацию в повседневную деятельность разработчиков**

Наша цель — обеспечить ориентированные на рынок результаты, чтобы много небольших команд могли быстро и независимо друг от друга доставлять продукт клиентам. Достижение цели может стать проблемой, если эксплуатация продукта осуществляется централизованно и функционально ориентированно, с обязательством удовлетворять потребности множества различных групп, а эти потребности потенциально могут быть абсолютно разными. Результатом часто оказывается длительное время выполнения задач эксплуатации, постоянные перераспределения приоритетов и обращения по этому поводу к руководству, а также неудачные решения по развертыванию.

Мы можем добиться ориентированных на рынок результатов путем интеграции функций эксплуатации в функции команд разработчиков, сделав эти команды более эффективными и продуктивными. В этой главе мы рассмотрим различные пути достижения этой цели как на организационном уровне, так и при помощи соответствующих повседневных действий. Тем самым отдел эксплуатации может значительно повысить производительность команд разработчиков во всей компании, а также обеспечить более тесное сотрудничество и лучшие результаты, получаемые организацией.

В компании Big Fish Games, которая разрабатывает и поддерживает сотни мобильных игр и тысячи игр для ПК и получила в 2013 г. доход свыше 266 миллионов долларов, вице-президент по ИТ-эксплуатации Пол Фаррел руководил соответствующим отделом. Он был ответственным за поддержку различных бизнес-подразделений, имевших значительную автономию.

Каждое из подразделений имело прикрепленные команды разработчиков, нередко выбирающие разные технологии. Когда эти команды хотели развернуть новые функциональные возможности, им приходилось конкурировать за общий пул дефицитных ресурсов отдела эксплуатации. Кроме того, им приходилось бороться с ненадежностью тестирования и интеграции сред, а также с крайне громоздкими процессами релиза.

Фаррел подумал, что наилучший способ решения этой проблемы — добавление знаний и навыков сотрудников отдела эксплуатации к знаниям разработчиков. Он отметил: «Когда у команд разработчиков были проблемы с тестированием или развертыванием, они нуждались в чем-то большем, чем просто технологии или окружение. Им нужны были помочь и наставничество. Сначала мы добавили специалистов по эксплуатации и архитекторов в каждую команду разработчиков, но у нас просто было недостаточно инженеров, чтобы охватить все команды. Мы смогли помочь разработчикам, применив то, что мы называли “контакт с отделом эксплуатации”, привлекая даже меньше людей».

Фаррел определил два типа «контактов с отделом эксплуатации»: менеджер по связям с бизнесом и прикрепленный инженер по релизу. Менеджеры по связям с бизнесом взаимодействовали с менеджерами продуктов, владельцами направлений бизнеса, проект-менеджерами, менеджерами команд разработчиков и собственно разработчиками. Они хорошо изучили бизнес-стимулы продуктовых групп и планы дальнейших разработок, выступали защитниками владельцев продуктов в отделе эксплуатации и помогали продуктовым командам ориентироваться в ИТ-эксплуатации при определении приоритетов и рационализации задач.

Точно так же прикрепленный инженер по релизу получает хорошее знание особенностей разработки и тестирования продукта и помогает команде тем, что нужно от отдела эксплуатации для достижения целей. Он хорошо знаком с типичными запросами разработчиков и тестировщиков в отдел эксплуатации и нередко оказывается в состоянии выполнить необходимую работу самостоятельно. По мере необходимости он будет также вовлекать специализированных инженеров эксплуатации (например, администраторов баз данных, специалистов по информационной безопасности, инженеров хранения данных, сетевых инженеров) и помогать определить, какие из инструментов для автоматизации наиболее распространенных запросов отдел эксплуатации в целом должен создавать в первую очередь.

Поступив таким образом, Фаррел смог помочь всем командам разработчиков в организации стать более продуктивными и достичь командных целей. Более того, он помог командам определить приоритеты с учетом глобальных ограничений, наложенных отделом эксплуатации, сократив тем самым количество неприятных сюрпризов, обнаруживаемых на середине проекта, и максимально увеличив общую производительность.

Фаррел отмечает благодаря этим изменениям увеличилась скорость выпуска новых релизов, а также улучшились рабочие отношения с отделом эксплуатации. Он заключает: «модель “контактов с эксплуатацией” позволила нам внедрить знания и навыки отдела эксплуатации в команды разработчиков продуктов без добавления нового персонала».

Преобразования DevOps в компании Big Fish Games показывают, как централизованная команда

эксплуатации смогла добиться результатов, обычно ассоциирующихся с рыночно ориентированными командами. Мы можем использовать следующие три общие стратегические установки:

- создать возможности самообслуживания, позволяющие разработчикам из сервисных команд действовать продуктивно;
- включить инженеров эксплуатации в команды разработки сервисов;
- назначить «контакты с отделом эксплуатации» для команд разработки, если включение инженеров в их состав не представляется возможным.

И наконец, опишем, как инженеры эксплуатации могут быть вовлечены в стандартные действия команды разработчиков при выполнении повседневной работы, в том числе ежедневных обсуждений, планирования и ретроспективных обзоров.

Один из способов формирования рыночно ориентированных решений — создание централизованных платформ и сервисов по предоставлению разного рода инструментов для решения задач эксплуатации, которые любая команда разработчиков сможет использовать, чтобы стать более продуктивной. Это предоставление среды, близкой к производственной, конвейеры развертывания, автоматизированные средства тестирования, панель с информацией о встроенной в продукт телеметрии и т. д.. Поступая таким образом, мы даем возможность командам разработчиков тратить больше времени на создание функциональных возможностей для клиентов, а не расходовать его на получение инфраструктуры, необходимой для включения этой новой возможности в производственную среду и ее поддержки.

Все предоставляемые нами платформы и сервисы должны в идеале быть автоматизированы и доступны по запросу, без необходимости для разработчиков создавать задачу и дожидаться, когда кто-нибудь выполнит ее вручную. Это гарантирует, что подразделение эксплуатации не станет узким местом для внутренних клиентов (например, «мы получили ваше задание, и его выполнение потребует шести недель на настройку тестовой среды вручную»).

Действуя так, мы дали командам возможность получать то, что им нужно, именно тогда, когда им это нужно, а также снизили потребность в коммуникациях и координации. Как отметил Дэймон Эдвардс, «без платформ с возможностью самообслуживания облако становится просто “Дорогим хостингом 2.0”».

Почти во всех случаях мы не накладываем на внутренние команды обязательств использовать эти платформы и сервисы — команды разработки этих платформ должны завоевывать внутренних клиентов, иногда даже конкурируя с внешними поставщиками. Благодаря созданию эффективного внутреннего рынка возможностей мы можем быть уверены, что платформы и сервисы, созданные нами, являются самыми простыми и привлекательными вариантами из доступных (путь наименьшего сопротивления).

Например, мы может создать платформу, предоставляющую репозиторий системы контроля версий хранилища данных с предварительно одобренными библиотеками безопасности, конвейер, автоматически запускающий инструменты проверки качества кода и безопасности и разворачивающий приложения в заведомо исправной среде с уже установленными инструментами мониторинга. В идеале мы можем сделать жизнь команд разработчиков настолько простой, что они будут в подавляющем большинстве случаев принимать решение, что платформа — самое простое, надежное и безопасное средство для передачи приложений в производственную среду.

Мы встраиваем в платформы накопленный коллективный опыт, полученный от каждого сотрудника в организации, в том числе из отделов тестирования, эксплуатации и подразделения информационной безопасности, что помогает создать безопасную систему. Это повышает продуктивность разработчиков и помогает командам улучшать общие процессы — например, выполнение автоматизированного тестирования и проверок соответствия требованиям безопасности и нормативным документам.

Создание и поддержание платформ и инструментов — реальная разработка продуктов, только клиентами платформы становятся не внешние потребители, а внутренние команды разработчиков. Как и создание любого отличного продукта, создание отличной, всеми любимой платформы не может произойти по воле случая. Внутренняя команда платформы с плохой нацеленностью на клиентов, скорее всего, создаст инструменты, вызывающие общую неприязнь. От них быстро откажутся в пользу других, будь то иная внутренняя платформа или продукт внешнего поставщика.

Диана Марш, директор по разработке инструментов компании Netflix, заявляет, что

основополагающим принципом в работе ее команды является «поддержка инноваций инженерных команд и скорости их работы. Мы ничего не создаем, не готовим, не развертываем для этих команд, мы не управляем их конфигурациями. Вместо этого мы создаем инструменты для организации самообслуживания. Инженеры зависят от наших инструментов, но важно, что они не попадают в зависимость от нас».

Часто эти платформенные команды предоставляют другие сервисы, чтобы помочь своим клиентам изучить технологии, или помогают осуществить миграцию других технологий. Обеспечиваются наставничество и консультации для улучшения методов работы организации. Общие сервисы также облегчают стандартизацию, позволяющую инженерам быстро становиться продуктивными, даже если они переходят в другую команду. Например, если каждая команда выбирает свой набор инструментов, то от инженеров может потребоваться изучение совершенно нового набора технологий, что ставит цели команды выше целей организации.

В организациях, где команды могут использовать только утвержденные инструменты, мы можем начать с отмены этого требования для нескольких команд, чтобы мы могли экспериментировать и открывать возможности сделать их более продуктивными.

Команды, включенные в разработку множества сервисов, должны постоянно изучать внутренние наборы инструментов, широко применяющиеся в организации, и решать, каким инструментам стоит уделить внимание и осуществить их централизованную поддержку, чтобы сделать их доступными для всех. В целом взять инструмент, который уже где-то применяется, и расширить сферу его использования — такой подход имеет гораздо больше шансов на успех, чем создание с нуля.

Другой способ получить рыночно ориентированные результаты — дать командам возможность стать более самодостаточными, пригласив инженеров эксплуатации и тем самым уменьшив зависимость от централизованного отдела эксплуатации. Эти команды могут также быть полностью ответственными за предоставление услуг и поддержку.

При включении инженеров эксплуатации в команды разработчиков приоритеты будут определяться почти исключительно целями команд. Напротив, цель отдела эксплуатации — преимущественно решение его собственных проблем. В результате инженеры становятся более тесно связаны с внутренними и внешними клиентами. Более того, у команд нередко достаточный бюджет, чтобы оплатить наем инженеров, хотя собеседование и решение о найме будут, скорее всего, оставаться за централизованным отделом эксплуатации для обеспечения качества персонала и однородности его состава.

Джейсон Сокс говорит: «В компании Disney есть системные инженеры, введенные в состав команд подразделений, в частности в отделы разработки, тестирования и даже информационной безопасности. Это полностью изменило динамику. В качестве инженеров эксплуатации мы создаем инструменты и возможности, преобразующие задачи и способы мышления. При традиционном подходе мы просто управляем поездом, сделанным кем-то другим. Однако при современной эксплуатации с помощью закрепленных инженеров мы помогаем его делать, причем не только сам поезд, но даже мосты, по которым он катится».

Начиная новый крупный проект разработки продукта, мы первоначально вводили в состав команд инженеров эксплуатации. Их деятельность может включать в себя помочь в решении, что надо делать и как. Они могут влиять на архитектуру продукта, помогая принимать решения по использованию тех или иных внутренних и внешних технологий, содействуя в создании новых возможностей в наших внутренних платформах и, возможно, даже создавая новые операционные возможности. После того как продукт запущен в производство, включенные в команду инженеры эксплуатации могут помочь решать проблемы, связанные с ответственностью команды разработчиков за производство.

Они будут принимать участие во всех действиях команды разработчиков, таких как планирование, ежедневные обсуждения и демонстрации, на которых группа показывает новые функциональные возможности и решает, какие передавать в производство. По мере того как необходимость в знаниях и способностях инженеров уменьшается, они могут переключиться на другие проекты или задания, следуя типовой схеме изменений команд в ходе жизненного цикла.

Парадигма имеет еще одно важное преимущество: объединение в пары разработчиков и инженеров эксплуатации — чрезвычайно эффективный способ кросс-профессионального обучения и накопления опыта в сервисных командах. Это может также дать значительную выгоду в виде преобразования знаний об эксплуатации в автоматизированный код, который может стать намного более надежным и широко использоваться повторно.

По ряду причин (например, стоимость персонала и его нехватка) мы можем оказаться не в состоянии включить инженера эксплуатации в каждую продуктовую команду. Однако мы по-прежнему можем получить множество преимуществ, назначая в каждую продуктовую команду

«контакт с эксплуатацией».

В компании Etsy такая модель получила название «выделенный инженер». Централизованная группа эксплуатации продолжает управлять всеми средствами — не только производственными, но также и предпроизводственными, чтобы обеспечить согласованность. «Выделенный инженер» отвечает за взаимопонимание относительно следующего:

- каковы новые функциональные возможности продукта и почему мы создаем его;
- каковы принципы его действия — в том, что касается работоспособности, масштабируемости и наблюдаемости (схематическое отображение настоятельно рекомендуется);
- каким образом отслеживать и собирать телеметрию для осуществления наблюдения за прогрессом и определения того, успешна или нет новая возможность;
- имеются ли какие-либо отступления от ранее использовавшихся архитектур и методик и как они обосновываются;
- имеются ли дополнительные потребности в инфраструктуре, и каким образом они будут влиять на общие функциональные возможности инфраструктуры;
- каковы планы запуска функций в производство.

Кроме того, как и в случае с включенными в команды инженерами, «контакт с эксплуатацией» присутствует на обсуждениях в команде, отвечает за учет потребностей в плане работы отдела эксплуатации и выполняет все необходимые задачи. Мы полагаемся на эти «контакты», если необходимо передать на решение руководства любые разногласия или решение вопросов о приоритетности задач. Поступая так, мы устанавливаем, какие ресурсы или конфликты, связанные с распределением времени, должны оцениваться (с определением приоритетов) в контексте более широких организационных целей.

Назначение «контактов с эксплуатацией» позволяет обеспечивать поддержку большего числа продуктовых команд, чем включение инженеров в команды. Цель в том, чтобы отдел не создавал ограничений для продуктовых групп. Если мы обнаружим, что «контакты с эксплуатацией» находятся на пределе возможностей, не позволяя командам достичь целей, то, скорее всего, нам придется или уменьшить количество команд, сопровождающих каждый «контакт», или временно включить инженера эксплуатации в состав конкретной команды.

Когда инженеры эксплуатации введены в состав групп или становятся «выделенными контактами» для общения с разработкой, мы можем включить их в процедуры команды разработчиков. В этом разделе цель — помочь инженерам эксплуатации и другим специалистам, не разработчикам, лучше понять существующую культуру разработки и активно интегрироваться во все аспекты планирования и повседневной деятельности. В результате отдел эксплуатации сможет лучше планировать работу и активнее передавать в продуктовые команды все необходимые им знания, оказывая влияние на работу задолго до того, как она попадет в производственную среду. В следующих разделах описаны некоторые стандартные процедуры, используемые командами разработчиков при обращении к методам Agile, и то, как мы должны вводить инженеров эксплуатации в эти команды. Внедрение методов Agile ни в коем случае не может стать предварительным требованием, наша цель — узнать, каким процедурам следуют продуктовые команды, встроиться в эти процедуры и увеличивать их ценность.

Как заметил Эрнест Мюллер, «я считаю, что DevOps работает намного лучше, если команды эксплуатации примут такие же гибкие процедуры, которые используют команды разработчиков. Мы уже добились фантастических успехов в решении многих проблем, связанных с болевыми точками эксплуатации и интеграции с командами разработчиков».

Одна из ежедневных процедур у разработчиков, популяризованная Scrum, — это ежедневное собрание, короткая встреча. На нее собирается вся команда, и до всеобщего сведения доводятся три вещи: что было сделано вчера, что будет сделано сегодня, что мешает завершить работу.

Цель этой процедуры — распространить свою информацию на всю команду и понять, какая работа уже сделана, а какую надо сделать. Когда члены команды представляют эту информацию друг другу, мы узнаем обо всех задачах, на пути разработки которых возникли препятствия, и найдем способы помочь друг другу продвинуть работу к завершению. Кроме того, если на собрании присутствуют менеджеры, то мы сможем быстро разрешить конфликты приоритизации и использования ресурсов.

Одна из обычных проблем — раздробленность информации между членами команды разработчиков. За счет участия в собраниях инженеров эксплуатации этот отдел может получить представление о деятельности команды, что обеспечивает эффективность планирования и подготовки. Например, если мы увидим, что команда планирует развертывание большой новой функциональности в течение следующих двух недель, то мы можем выделить на поддержку развертывания нужный персонал и достаточное количество ресурсов. Или, наоборот, можем выявить области, где необходимо более тесное взаимодействие или дополнительная подготовка (например, создание большего количества мониторинговых проверок или автоматизированных сценариев). Сделав это, мы создадим условия, при которых отдел эксплуатации сможет помочь в решении проблем данной команды (например, повысив производительность путем настройки базы данных вместо оптимизации кода) или проблем, возможных в будущем, до того как они станут серьезными (например, обеспечив более тесную интеграцию сред тестирования, чтобы сделать возможным тестирование производительности).

Другая широко распространенная процедура Agile — ретроспективные обзоры. В конце каждого этапа разработки группа обсуждает, что сделано успешно, что можно улучшить и как включить успехи и улучшения в будущие итерации или проекты. Команда предлагает идеи, как сделать работу лучше, и рассматривает результаты экспериментов, проведенных на предыдущей итерации. Это один из основных механизмов организационного обучения и разработки контрмер, а полученные результаты работ немедленно реализуются или вносятся в список незавершенных дел команды.

Если инженеры эксплуатации участвуют в ретроспективных обзорах проектной команды, они также могут выиграть от приобретения новых знаний. Кроме того, если на этом временном интервале выполняется развертывание или релиз, инженеры должны рассказать о результатах и любых полученных выводах, создавая обратную связь с продуктовой командой. Поступая так, мы можем оптимизировать планирование будущей работы и ее выполнение, улучшая результаты. Приведем примеры обратной связи, которую ИТ-инженеры могут привнести в ретроспективный обзор.

- «Две недели назад мы обнаружили слепое пятно, которое не отслеживалось системой мониторинга, и пришли к согласию, как это исправить. Наше решение заработало. В прошлый вторник у нас произошел сбой, но мы быстро обнаружили его и исправили еще до того, как он затронул клиентов».
- «На прошлой неделе развертывание оказалось одним из наиболее трудных и продолжительных более чем за год. Вот некоторые соображения, каким образом его можно улучшить».
- «Маркетинговая кампания, которую мы проводили на прошлой неделе, была гораздо труднее, чем мы ожидали, и мы, вероятно, не должны больше делать такие предложения. Вот некоторые идеи по другим предложениям, которые мы можем сделать для достижения наших целей».
- «В ходе последнего развертывания самой большой нашей проблемой были правила брандмауэра, в настоящее время состоящие из тысяч строк, так что изменять их чрезвычайно трудно и опасно. Мы должны перепроектировать систему предотвращения несанкционированного сетевого трафика».

Обратная связь от эксплуатации помогает командам лучше увидеть и понять последствия принятых решений для производственной среды. Если результаты отрицательные, мы можем внести необходимые изменения, чтобы не повторять ошибок в будущем. Эта обратная связь также, скорее всего, поможет выявить больше проблем и дефектов, подлежащих исправлению, она даже может выявить более крупные архитектурные проблемы.

Дополнительная необходимая работа, выявленная в ходе ретроспективного обзора проектной команды, относится к широкой категории работ по улучшению, таких как устранение дефектов, рефакторинг и автоматизация ручных действий. Менеджеры продуктов и проектов могут захотеть отложить такие работы по улучшению или понизить их приоритет в пользу работ над новыми функциями для клиентов.

Однако мы должны напомнить, что улучшение повседневной работы важнее, чем работа сама по себе, и что все команды должны иметь выделенные ресурсы (например, резервирование 20 % всего времени на улучшение работы, для чего нужно запланировать выделение одного дня в неделю или одной недели в месяц и так далее). Без этого продуктивность команды будет почти наверняка значительно снижаться под давлением собственного технического долга.

Зачастую команды разработчиков делают свою работу видимой с помощью проектных досок или досок канбан. Однако намного реже на досках отображаются работы, которые отдел эксплуатации должен выполнить, чтобы приложение было успешно запущено в производство, где фактически

создается продукт для клиента. В результате мы не знаем о необходимой работе, пока она не оказывается нужна позарез, ставя под угрозу сроки выполнения проекта или грозя производственным простоем.

Поскольку эксплуатация — часть потока создания ценности продукта, мы должны показать работу, выполняемую этим отделом и связанную с доставкой продукта, на общей доске канбан. Это позволит нам более четко увидеть все работы, необходимые для передачи нашего кода в производство, а также отслеживать все действия отдела эксплуатации, необходимые для поддержки продукта. Кроме того, мы сможем увидеть, где работа этого отдела блокируется, где необходимо передать проблему на рассмотрение руководства, а также выявятся области, которые нужно усовершенствовать.

Доска канбан — идеальное средство сделать задачи наглядно видимыми, а наглядность — ключевой компонент в том, чтобы должным образом признать вклад отдела эксплуатации и интегрировать его работу во все соответствующие потоки создания ценности. Когда мы делаем это хорошо, мы можем достичь рыночно ориентированных решений независимо от того, какова организационная схема компании.

В этой главе мы рассмотрели способы интеграции отдела оперирования в повседневную работу разработчиков и то, как сделать нашу работу более видимой для этого отдела. Мы изучили три стратегии, с помощью которых можно довести это дело до конца, в том числе создание возможностей самообслуживания, позволяющих разработчикам в сервисных командах работать продуктивно, включение инженеров эксплуатации в сервисные команды и «контактов с эксплуатацией». И наконец мы описали, как инженеры эксплуатации могут интегрироваться в команды разработчиков, включаясь в их повседневную работу, в том числе участвуя в ежедневных собраниях, в планировании и в ретроспективных обзорах.

В части II мы изучили различные способы обдумывания преобразований DevOps, в том числе как выбрать место, где начинать преобразования, обсудили соответствующие аспекты архитектуры и организационного проектирования и то, как организовать наши команды. Мы также изучили, как интегрировать сотрудников отдела эксплуатации во все аспекты планирования разработок и в повседневную деятельность разработчиков.

В части III начнем изучать вопрос, как внедрить конкретные технические методы для реализации принципов потока, позволяющие обеспечить быстрое течение работы от разработчиков к отделу эксплуатации, не вызывая хаоса и срывов на производстве.

## **Введение**

В части III наша цель — описать технические методы и архитектуру, необходимые для создания и поддержания быстрого потока задач от разработчиков к эксплуатации без хаоса и сбоев в производственной среде или у клиентов. Нам требуется уменьшить риски, связанные с развертыванием и релизом изменений в производство. Мы будем делать это, реализовывая комплекс технических методов, известный как *непрерывная поставка*.

Непрерывная поставка включает в себя создание основ конвейера автоматизированного развертывания, обеспечение автоматизированного тестирования, которое постоянно напоминает, что мы находимся в состоянии пригодности к развертыванию, а также включает ежедневную интеграцию разработчиками своего кода в основную ветвь разработки и проектирование сред и кода для обеспечения релиза продукции с низкими рисками. Основное внимание в этих главах уделяется:

- созданию основы нашего конвейера развертывания;
- быстрому и надежному автоматизированному тестированию;
- организации непрерывных интеграции и тестирования;
- обеспечению возможности низкорисковых релизов, их проектирование и автоматизация.

Реализация этих методов сокращает время получения сред, близких к производственным, обеспечивает непрерывное тестирование, предоставляет каждому быструю обратную связь о его работе, позволяет небольшим командам безопасно и самостоятельно разрабатывать, тестировать и развертывать их код в эксплуатацию и делает релизы продукта и его развертывание в производственной среде привычной частью повседневной работы.

Кроме того, интеграция целей тестировщиков и эксплуатации в повседневной работе снижает количество неотложной работы, затруднений и саму напряженность работы, в то же время она делает сотрудников более продуктивными и увеличивает их удовлетворенность работой. Мы не только усиливаем отдачу от работы, но наша организация сможет легче добиваться успеха на рынке.

## **Глава 9. Создание основы конвейера внедрения**

Чтобы создать быстрый и надежный поток от разработчиков к эксплуатации, мы должны обеспечить использование сред, близких к производственным, на каждой стадии потока создания ценности. Кроме того, эти среды должны создаваться автоматизированно, в идеальном случае — по требованию, с помощью сценариев и информации о конфигурациях, хранящихся в системе контроля версий, среды должны обслуживаться полностью автоматически, не требуя каких-либо работ вручную, выполняемых отделом эксплуатации. Наша цель — обеспечить создание заново всей производственной среды, основываясь на данных, хранящихся в системе контроля версий.

Слишком часто мы обнаруживаем, как наши приложения работают в среде, приближенной к производственной, только во время развертывания — когда уже слишком поздно, чтобы устранять проблемы без негативного влияния на клиентов. Наглядный пример широкого спектра проблем, вызванных рассогласованием приложений и сред, — программа Enterprise Data Warehouse. Ее созданием в крупной австралийской телекоммуникационной компании в 2009 г. руководила Эм Кэмпбел-Претти. Она стала генеральным менеджером и бизнес-спонсором этой программы, стоившей 200 миллионов долларов, и одновременно получила ответственность за все стратегические цели, связанные с этой платформой.

В презентации на конференции DevOps Enterprise Summit в 2014 г. Кэмпбел-Претти пояснила, что «в то время параллельно шли десять потоков работы, все с помощью водопадного метода, и все десять потоков значительно отставали от графика. Только один из потоков успешно дошел до стадии приемочных испытаний, проводимых пользователями (User Acceptance Testing, UAT), в запланированный срок, и еще шесть месяцев потребовались для завершения этих испытаний, которые показали, что результирующая производительность оказалась намного ниже ожидаемой. Эта недостаточная производительность явилась основным катализатором преобразования отдела по технологии Agile».

Однако после использования Agile в течение почти года разработчики добились только небольших улучшений, по-прежнему не получая необходимых результатов в бизнесе. Кэмпбел-Претти провела ретроспективный обзор программы и задала команде вопрос: «Если учесть весь опыт, накопленный нами за время последнего релиза, то какие вещи мы могли бы сделать, чтобы удвоить нашу продуктивность?»

В ходе проекта раздавалось ворчание об «отсутствии пользы для бизнеса». Однако в ходе ретроспективного обзора тезис «улучшить доступность сред» оказался в начале списка проблем. Ретроспективный подход ясно показал: командам разработчиков для начала необходимо получить соответствующую среду, а время ее ожидания нередко растягивалось до восьми недель.

Новая команда интеграции и сборки стала ответственной за «создание качества внутри процессов вместо проверки качества после создания продукта». Первоначально она была в составе команды администраторов баз данных (DBA), и специалисты получили задачу по автоматизации процесса создания среды. Команда быстро сделала удивительное открытие: только 50 % исходного кода в средах разработки и тестирования соответствовало производственной среде.

Кэмпбел-Претти отмечала: «Неожиданно мы поняли, почему каждый раз при развертывании нашего кода в новых средах сталкивались с таким количеством дефектов. В каждой среде мы продвигались вперед с исправлением ошибок, но сделанные изменения не попадали в хранилище контроля версий».

Команда тщательно выполнила реверсивное проектирование всех изменений, внесенных в различные среды, и зафиксировала их в хранилище контроля версий. Она также автоматизировала процесс создания сред, с тем чтобы можно было создавать их повторно и правильно.

Кэмпбел-Претти описала результаты, отмечая: «время, которое теперь требовалось для получения правильной среды, уменьшилось с восьми недель до одного дня. Это оказалось одним из ключевых изменений, позволивших нам достичь целей в отношении времени разработки, стоимости доставки и количества дефектов в производстве, которых нам удалось избежать».

История, рассказанная Кэмпбел-Претти, демонстрирует разнообразие проблем в системе, где использовались несовместимые среды и систематически не вносились изменения в хранилище контроля версий.

На протяжении остальной части этой главы мы будем обсуждать, как создать механизмы, позволяющие моделировать среды по требованию, расширить использование контроля версий на каждого включенного в поток создания ценности, добиться, чтобы инфраструктуру было проще построить заново, нежели восстанавливать, и обеспечить, чтобы разработчики запускали свой код в средах, приближенных к производственным, на каждом этапе жизненного цикла программного обеспечения.

Как можно видеть из приведенных выше примеров корпоративных центров обработки данных, одна из основных причин хаоса, дезорганизации и иногда даже катастрофических последствий релиза программного обеспечения — то, что, к сожалению, только в процессе релиза мы впервые узнаем, как приложение работает в среде с реальными нагрузками и в реальной производственной среде. Во многих случаях команды разработчиков могут получить запрошенную тестовую среду на ранних этапах осуществления проекта.

Однако, когда отделу эксплуатации требуется длительное время для подготовки тестовых сред по заказам для проведения необходимого тестирования, команды не могут получить их достаточно быстро. Что еще хуже, тестовые среды часто неправильно настроены или настолько отличаются от производственных сред, что мы до сих пор сталкиваемся с большими проблемами на производстве, несмотря на проведенное перед развертыванием тестирование.

На этом этапе мы хотим, чтобы разработчики использовали среду, близкую к производственной, созданную по их требованию и самообслуживающуюся. При этом разработчики могут запускать и тестировать код в среде, близкой к производственной, как часть повседневной работы, обеспечивая тем самым получение ранней и постоянной обратной связи о качестве своей работы.

Вместо того чтобы просто описывать характеристики производственной среды в документе или на странице wiki, мы создаем механизм, создающий все наши среды, в том числе для разработки, тестирования и производства. При этом любой член команды может получить среду, близкую к производственной, через несколько минут без необходимости создавать задачу на выполнение работы, не говоря уже о необходимости ждать неделями.

Чтобы это реализовать, надо иметь описания и автоматизировать создание проверенных, заведомо исправных сред, стабильных, безопасных: риск их использования невелик, они аккумулируют коллективные знания организации. Все требования к средам изложены не в документах, и знания о них не хранятся в чьей-либо голове. Они кодифицированы в процессе автоматизированного создания сред.

Вместо того чтобы отдел эксплуатации вручную создавал и настраивал среду, мы можем использовать автоматизацию для следующих способов создания среды:

- копирование виртуализированной среды (например, образа VMware, запуск сценария Vagrant, загрузка файла Amazon Machine Image в EC2);
- создание процесса автоматизированного формирования среды, который начинает работу с нуля (например, PXE — установка из базовых образов);
- с помощью инструментов управления конфигурациями «инфраструктура как код» (например, Puppet, Chef, Ansible, Salt, CFEngine и так далее);
- с помощью автоматизированных инструментов конфигурации операционной системы (например, Solaris Jumpstart, Red Hat Kickstart, Debian preseed);
- сборка среды из набора виртуальных образов или контейнеров (например, Vagrant, Docker);
- сборка новой среды в общедоступной среде облачных вычислений (например, Amazon Web Services, Google App Engine, Microsoft Azure), частном облаке или в других PaaS (таких, как OpenStack или Cloud Foundry и так далее).

Поскольку мы тщательно определили все аспекты заблаговременного создания среды, мы не только в состоянии создавать новые среды быстро, но также и обеспечиваем их стабильность, надежность и безопасность. От этого выигрывают все.

Отдел эксплуатации получает выгоду от возможности быстрого создания новых сред, потому что автоматизация процесса улучшает согласованность сред и уменьшает количество утомительной, подверженной ошибкам работы вручную. Кроме того, разработчики также получают выгоду, поскольку оказываются в состоянии воспроизвести все необходимые детали в производственной среде для создания, запуска и проверки их кода на своих рабочих станциях. Тем самым мы даем разработчикам возможность найти и устранить многие проблемы еще на ранних стадиях осуществления проекта, а не в ходе интеграционного тестирования или, что еще хуже, после релиза в производство.

Предоставляя разработчикам полностью управляемую среду, мы даем им возможность быстро воспроизводить, диагностировать и устранять дефекты, причем их работа надежно изолирована от производственных серверов и других общих ресурсов. Разработчики могут экспериментировать с

изменениями в средах, а также с кодом инфраструктуры, который их создает (например, сценариями Configuration Management), продолжая создавать новые знания, общие для разработки и IT-отдела.

На предыдущем шаге мы сделали возможным создание сред разработки, тестирования и производства по запросу. Теперь надо заняться остальными частями программной системы.

Десятилетиями всеобъемлющее использование контроля версий все активнее становилось обязательной практикой как индивидуальных разработчиков, так и команд. Система контроля версий записывает изменения в файлах или наборах файлов, хранящихся внутри системы. Это может быть исходный код, другие активы или любые документы как часть проекта создания программного обеспечения. Мы вносим изменения в группы, называемые коммитами, или редакциями. Каждый коммит вместе с метаданными (кто внес изменения и когда) сохраняется в системе тем или иным способом, что позволяет нам сохранять, сравнивать, выполнять слияния и восстанавливать предыдущие версии объектов в репозитории. Это решение также минимизирует риски, поскольку предоставляет способ откатить объекты, измененные в производственной среде, к предыдущим версиям (в дальнейшем следующие термины будут использоваться как равнозначные: загрузить в систему контроля версий, зафиксировать в системе контроля версий, зафиксировать код, зафиксировать изменения, фиксация, коммит).

Когда разработчики помещают в систему контроля версий все файлы исходного кода и файлы конфигураций, она становится единственным хранилищем истины и содержит точные состояния, предназначающиеся для использования. Однако поскольку для доставки продукта клиенту требуется и наш код, и среда, в которой он работает, надо, чтобы среда также хранилась в системе контроля версий. Другими словами, контроль версий должен использоваться каждым человеком в нашем потоке создания ценности, включая тестировщиков, отделы IT и информационной безопасности, а также разработчиков. Если все объекты, связанные с производством, помещены в систему контроля версий, репозиторий этой системы дает нам возможность неоднократно и достоверно воспроизводить все компоненты нашей рабочей системы программного обеспечения — это включает в себя наши приложения и производственную среду, равно как и все наши предпроизводственные среды.

Чтобы мы могли восстановить производственные сервисы с повторяемостью и предсказуемо (и в идеале быстро) даже при катастрофических событиях, мы должны проверить, хранятся ли в репозитории системы хранения версий следующие активы:

- весь код приложения и все зависимости (например, библиотеки, статическое содержимое и так далее);
- все сценарии, использующиеся для создания схем баз данных, справочные данные приложений и так далее;
- все инструменты для создания среды и артефактов, описанных на предыдущем шаге (например, образы VMware или AMI, рецепты Puppet или Chef и так далее);
- все файлы, использующиеся для создания контейнеров (например, определения или композиционные файлы состава Docker или Rocket);
- все вспомогательные автоматические тесты и все сценарии тестирования вручную;
- все сценарии, обеспечивающие упаковку кода, развертывание, миграцию баз данных и предоставление рабочей среды;
- все артефакты проекта (например, документации с описанием требований к продукту, процедуры развертывания, примечания к релизу и так далее);
- все файлы конфигурации облаков (например, шаблоны формирования AWS Cloud, файлы Microsoft Azure Stack DSC, OpenStack HEAT);
- все другие сценарии или информация о конфигурации, требующаяся для создания инфраструктуры, которая поддерживает несколько служб (например, шина служб предприятия, системы управления базами данных, файлы конфигурации зоны DNS, правила для межсетевых экранов и других сетевых устройств).

Можно иметь несколько репозиториев для различных типов объектов, в которых они маркированы и помечены наравне с исходным кодом. Например, мы можем хранить большие образы виртуальных машин и ISO-файлы, собранные двоичные файлы и тому подобное в репозиториях артефактов

(таких, как Nexus, Artifactory и так далее). Или можем положить их в хранилища blob (например, блоки Amazon S3) или положить образы Docker в хранилища Docker и так далее.

Но недостаточно просто иметь возможность заново воссоздать любое из предыдущих состояний производственной среды; мы должны также иметь возможность воссоздать целиком предпроизводственную среду и процессы сборки. Поэтому нам необходимо включить в систему контроля версий все, что используется в ходе сборки, в том числе инструменты (например, компиляторы и инструменты для тестирования) и среды, от которых они зависят.

Согласно докладу 2014 State of DevOps Report компании Puppet Labs использование отделом эксплуатации системы контроля версий — наилучший показатель для предсказывания производительности как ИТ-подразделений, так и всей организации в целом. По сути, этот показатель даже лучше, чем использование системы контроля версий разработчиками.

Выводы из этого доклада подчеркивают важнейшую роль, которую система контроля версий играет в процессе разработки программного обеспечения. Теперь мы знаем: если все приложения и все изменения среды записываются в системе контроля версий, мы можем не только быстро увидеть все изменения, приводящие к проблеме, но также обеспечить средства возврата к предыдущему состоянию, о котором нам известно, что оно рабочее. Это дает возможность быстрее справляться со сбоями.

Но почему использование системы контроля версий для сред разработки позволяет точнее предсказать производительность ИТ-подразделений и организации в целом, чем использование этой системы непосредственно для контроля кода?

Потому что почти во всех случаях количество конфигураций сред превосходит по порядку величины количество конфигураций нашего кода.

Контроль версий также предоставляет средства коммуникации для всех работающих в данном потоке создания ценности. Разработчики, тестировщики, сотрудники отделов эксплуатации и информационной безопасности могут видеть вносимые друг другом изменения, что помогает уменьшить количество неприятных сюрпризов, делает работу каждого прозрачной для всех и помогает создать и укрепить доверие (см. ).

Будучи способны по первому требованию быстро заново собрать или пересоздать приложения и среды, мы можем делать это вместо исправлений, когда что-то пошло не так. Хотя именно этим приемом пользуются практически все крупные (имеющие более тысячи серверов) веб-операторы, такую практику надо применять, даже если в производственной среде у нас работает только один сервер.

Билл Бейкер, всеми уважаемый инженер из компании Microsoft, язвительно заметил, что мы подходили к исправлению серверов как к лечению домашних питомцев: «Вы даете им имя, и если они заболевают, то ухаживаете за ними, пока они не выздоровеют. Сейчас с серверами обращаются как с крупным рогатым скотом. Вы нумеруете их, а если они заболевают, вы их пристреливаете».

Имея системы повторяемого воссоздания сред, мы можем легко увеличить мощность сервиса, добавляя дополнительные серверы в развертывание (выполняя горизонтальное масштабирование). Мы также можем избежать катастрофических последствий, которые непременно произойдут, если нам придется восстанавливать серверы после серьезного отказа невоспроизводимой инфраструктуры, сформировавшейся через несколько лет нездокументированных и вносившихся вручную изменений в производственной среде.

Обеспечивая согласованность сред независимо от изменений (изменения конфигурации, установка исправлений, модернизация и так далее), необходимо реплицировать их повсюду в производственных, предпроизводственных, а также в любых вновь создаваемых средах.

Вместо входа на серверы вручную и внесения изменений в конфигурацию мы должны внести изменения таким образом, чтобы все они автоматически реплицировались всюду и автоматически же фиксировались в системе контроля версий. Мы можем полагаться на нашу систему управления конфигурацией для обеспечения согласованности сред (например, Puppet, Chef, Ansible, Salt, Bosh и др.). Можно создавать новые виртуальные машины или контейнеры с помощью механизма автоматизированной сборки и развертывать их в производство, уничтожая старые машины или выводя их из ротации.

Последняя модель — так называемая постоянная инфраструктура: в ней внести изменения в производственную среду вручную невозможно, единственный способ — внести изменения в систему контроля версий и создать код и среду «с нуля». При таком подходе вариабельность никак не сможет «впопыхах» в производственную среду.

Во избежание неконтролируемых вариаций конфигурации мы можем отключить удаленный вход на производственные серверы или регулярно уничтожать экземпляры и заменять их новыми, чтобы

обеспечить удаление вместе с ними всех изменений, внесенных вручную. Такой подход мотивирует всех членов команд вносить изменения как положено, через систему контроля версий. Применяя такие меры, мы систематически уменьшаем возможность того, что инфраструктура отклонится от рабочего состояния (например, из-за смены конфигураций, повреждаемости отдельных компонентов, чьих-то сознательных усилий и так далее).

Также мы должны поддерживать в актуальном состоянии предпроизводственные среды — в частности, необходимо сделать так, чтобы разработчики максимально обновленный вариант среды. Разработчики часто стремятся работать в старых средах, поскольку не любят изменений: так можно нарушить работу имеющихся функций. Однако мы хотим обновлять среды часто, чтобы можно было обнаруживать проблемы на как можно более ранней стадии жизненного цикла проекта.

Теперь, когда наши среды могут быть созданы по требованию и все вносится в систему контроля версий, наша цель — обеспечить, чтобы в повседневной деятельности разработчиков использовались именно эти среды. Необходимо убедиться, что приложение работает в среде, приближенной к производственной, так, как и ожидалось, задолго до окончания проекта или до его первого развертывания в производстве.

Большинство современных методик разработки программного обеспечения предписывают короткие интервалы разработки и итеративность процесса в отличие от подхода big bang (например, модели водопада). В целом чем больше интервал между развертываниями, тем хуже результаты. Например, в методологии Scrum есть понятие *спринт* — некоторый интервал разработки фиксированной продолжительности (обычно один месяц или менее), в течение которого мы обязаны сделать то, что в широком смысле называется «работающий и потенциально готовый к поставке код».

Наша цель — обеспечить, чтобы разработчики и тестировщики регулярно интегрировали код в среду, близкой к производственной, с интервалами, постепенно сокращающимися по мере выполнения проекта. Мы делаем это путем расширения определения «выполнено» за границы правильного функционирования кода: в конце каждого интервала разработки мы должны иметь интегрированный, протестированный, работающий и потенциально готовый к поставке код, указанные характеристики которого **продемонстрированы в среде, близкой к производственной**.

Другими словами, мы будем считать задачу разработчиков выполненной, если она будет успешно собрана, развернута и подтверждется, что она работает, как и ожидалось, в среде, близкой к производственной, а не тогда, когда разработчик считает ее сделанной. В идеале она работает в среде, близкой к производственной, с набором данных, близким к производственному, задолго до окончания спрента. Это предотвращает ситуации, когда функцию называют сделанной лишь потому, что разработчик может успешно запустить ее на своем ноутбуке и нигде больше.

За счет того, что разработчики могут писать, тестировать и запускать код в среде, близкой к производственной, успешная интеграция кода и среды происходит во время повседневной работы, а не в конце релиза. К концу первого интервала можно продемонстрировать, что приложение правильно работает в среде, приближенной к реальной, что код и среда были интегрированы уже много раз, и в идеале полностью автоматизировано (никакой правки вручную не потребовалось).

Что еще лучше: к концу проекта мы успешно развернем и запустим код в средах, близких к производственным, сотни или даже тысячи раз, и это даст уверенность, что большинство проблем при развертывании продукции найдены и устранены.

В идеале мы можем использовать такие же инструменты — мониторинг, журналирование и развертывания — в предпроизводственных средах, как делаем это в производственных. Поступая так, мы приобретаем осведомленность и опыт, помогающие беспрепятственно развернуть и запустить сервис, когда он окажется в производственной среде, а также диагностировать проблемы и устраниить их.

Дав возможность разработчикам и группе эксплуатации приобрести общее знание того, как взаимодействуют код и среда, попрактиковаться в ранних и частых развертываниях, мы значительно снижаем риски развертывания, связанные с релизами кода продукта. Это также позволяет нам избавиться от целого класса дефектов эксплуатации и безопасности, архитектурных проблем, которые обычно проявляют себя, когда уже бывает слишком поздно, чтобы их исправлять.

Быстрый поток работы от разработчиков к группе эксплуатации подразумевает: любой работник по первому требованию может получить среду, приближенную к производственной. Дав разработчикам возможность использовать ее уже на самых ранних этапах проекта создания программного обеспечения, мы значительно снижаем риск появления производственных проблем впоследствии. Это один из многих методов, демонстрирующих, как отдел эксплуатации может сделать труд разработчиков гораздо более продуктивным. Мы закрепляем у разработчиков практику запускать код в условиях, близких к производственным, включая это условие в определение состояния «Закончено».

Более того, передавая производственные артефакты под управление системы контроля версий, мы получаем «единственный источник истины», что позволяет пересоздавать производственную среду быстро, повторяя и документируя, применяя к отделу управления те же методы работы, что и к разработчикам. А поскольку создать производственную структуру заново оказывается более легким делом, чем восстанавливать ее, решать проблемы удается быстрее и проще. Увеличивать мощность сервиса также становится легче.

Применение этих методов в правильных местах создает основу условий для всеобъемлющей автоматизации тестирования, которая рассматривается в следующей главе.

## **Глава 10. Быстрое и надежное автоматизированное тестирование**

На этом этапе разработчики и тестировщики используют в повседневной работе среды, приближенные к производственным. Мы успешно выполняем интеграционную сборку и запускаем код в такой среде после добавления каждой новой функции, при том что все изменения фиксируются в системе контроля версий. Однако мы, скорее всего, получим нежелательные результаты, если будем искать и исправлять ошибки на отдельном этапе тестирования, выполняемом отдельным подразделением уже после полного окончания разработки. И если тестирование выполняется только несколько раз в году, то разработчики узнают о допущенных промахах лишь через несколько месяцев после того, как внесли изменение, приведшее к ошибке. За это время связь между причиной и следствием будет, скорее всего, потеряна, а решение проблемы требует героических усилий и буквально археологических раскопок. Что самое плохое, значительно уменьшится наша способность учиться на ошибках и применять полученный опыт в будущей работе.

Автоматизированное тестирование решает еще одну серьезную и тревожащую проблему. Гэри Грувер отмечает: «Если нет автоматизированного тестирования, то чем больше кода мы пишем, тем больше времени и средств требуется для проверки, и в большинстве случаев это абсолютно немасштабируемая бизнес-модель для любой технологической организации».

Сейчас компания Google, несомненно, является примером внутренней производственной культуры, должным образом ценящей автоматизированное тестирование. Но такой подход соблюдался не всегда. В 2005 г., когда Майк Блэнд пришел на работу в компанию, развертывание обновлений сайта зачастую сопровождалось серьезными проблемами, особенно для команды Google Web Server (GWS).

Как объясняет Блэнд, «команда GWS попала в описанную выше ситуацию в середине 2000-х гг. Ей было чрезвычайно трудно внести изменения в веб-сервер — приложение на C++, обрабатывавшее все запросы к главной странице Google и многим другим веб-страницам сайта. При всей важности и известности работы в составе команды GWS была отнюдь не гламурным занятием — зачастую она напоминала поиски на свалке кода, реализующего различные функции, написанные командами, которые работали независимо друг от друга. Они сталкивались с такими проблемами, как слишком длительные сборка и тестирование кода, запуск в производство непротестированного кода, проходящая лишь изредка запись изменений кода, причем эти изменения вступали в противоречие с вносимыми другими командами».

Последствия всего этого были серьезными: результаты поиска могли содержать ошибки, или сам поиск был неприемлемо медленным, что влияло на тысячи поисковых запросов на сайте . Потенциальный результат — потеря не только дохода, но и доверия клиентов.

Блэнд описывает, как сумел повлиять на разработчиков, развертывавших изменения: «Страх стал убийцей мышления. Страх перед изменениями останавливал новых членов команды, они не понимали, как работает система. Но страх останавливал также и опытных сотрудников, так как они очень хорошо понимали последствия». Блэнд был частью группы, решавшей эту проблему.

Руководитель команды GWS Бхарат Медиратта считал, что автоматизированное тестирование поможет решить проблему. Как описывает Блэнд, «они утвердили жесткую позицию: изменения не будут приняты в GWS, если они не прошли автоматизированное тестирование. Они настроили непрерывную сборку и буквально с религиозным упорством соблюдали это правило. Они организовали отслеживание уровня тестового покрытия и обеспечивали постоянный его рост с течением времени. Они дописали политику и руководства по тестированию и настаивали, чтобы все участники, связанные с этими процессами как внутри команды, так и вне ее, строго соблюдали установленные правила».

Результаты поразили воображение. Как отмечает Блэнд, «GWS быстро стала одной из самых продуктивных команд в компании, выполняя интеграционную сборку большого числа изменений, поступающих от разных команд каждую неделю, поддерживая при этом график быстрых релизов. Новые члены команды теперь почти сразу же начинали плодотворно работать, внося вклад в сложную систему благодаря хорошему покрытию кода тестами и его качеству. В итоге радикальная политика позволила главной странице сайта быстро расширить возможности, преуспев в стремительно меняющемся мире конкурирующих технологий».

Но GWS все же была относительно небольшой командой в крупной и растущей компании. Команда хотела распространить применяемые методы на всю организацию. Поэтому на свет появилась Testing Grouplet, неофициальная группа инженеров, желавших распространить автоматизированное тестирование во всей организации. В течение следующих пяти лет они помогли растиражировать культуру автоматического тестирования на все подразделения компании Google.

Теперь, когда любой разработчик компании выполняет запись изменений кода, сразу запускается

набор из сотен тысяч автотестов. Если код проходит проверку, то он автоматически включается в основную ветку и оказывается готовым к развертыванию в производственной среде. Многие продукты Google собираются ежечасно или ежедневно, другие используют философию доставки Push on Green.

Ставки при таком подходе выше, чем когда бы то ни было: одна-единственная ошибка развертывания кода может одномоментно нарушить работу всего комплекса программ Google (например, из-за глобальных изменений в инфраструктуре или если дефект внесен в одну из основных библиотек, от которой зависит каждая программа).

Эран Мессери, инженер группы Google Developer Infrastructure, отмечает: «Время от времени случаются большие неудачи. Вы получаете кучу мгновенных сообщений, а инженеры стучатся в вашу дверь. Когда конвейер развертывания сломан, мы должны исправить это сразу, потому что разработчики не могут больше записывать изменения кода. Поэтому мы хотим сделать откат очень легким».

Эта система работает в компании Google благодаря профессионализму инженеров и культуре высокого доверия, предполагающей, что каждый хочет сделать свою работу хорошо и что мы можем быстро обнаруживать проблемы и исправлять их. Мессери объясняет: «В Google нет жестких правил, например “если вы вызовете остановку у более чем десяти проектов, то обязаны устранить проблему в течение десяти минут”. Вместо этого существует взаимное уважение между командами, подразумевается, что каждый делает все от него зависящее для поддержания конвейера развертывания. Все мы знаем, что однажды я могу нечаянно повредить ваш проект, а на следующий день вы можете сломать мой».

Результаты, полученные Майком Блэндом и командой Testing Grouplet, сделали компанию Google одной из самых продуктивных технологических организаций в мире. К 2013 г. автоматизированное тестирование и непрерывная интеграция позволили более чем 4000 независимых команд в компании работать вместе и оставаться продуктивными, одновременно выполняя разработку, непрерывную интеграцию, тестирование и развертывание своего кода в производственной среде. Весь код хранится в одном репозитории, он состоит из миллиардов файлов, и они постоянно используются для сборки и интеграции, причем ежемесячно код обновляется наполовину. Некоторые другие статистические данные об их производительности выглядят весьма внушительно:

- 40 000 записей изменений кода в день;
- 50 000 сборок в день (в выходные дни их число может превысить 90 000);
- 120 000 наборов автоматизированных тестов;
- 75 миллионов тестов выполняется ежедневно;
- свыше 100 инженеров, занимающихся разработкой тестов, непрерывной интеграцией и созданием инструментов для увеличения производительности труда разработчиков (что составляет 0,5 % от общего числа разработчиков).

В оставшейся части этой главы мы рассмотрим методы непрерывной интеграции, дающие возможность повторить эти результаты.

Наша цель — обеспечить качество нашего продукта уже на самых ранних этапах, а для этого разработчики должны организовать автоматизированное тестирование как часть их повседневной работы. Это создает быструю обратную связь, что помогает разработчикам рано обнаруживать проблемы и быстро их исправлять, пока еще это не требует серьезных затрат (например, времени и ресурсов).

На этом этапе мы создаем наборы автоматических тестов, обеспечивающие увеличение частоты интеграций и превращающие тестирование нашего кода и наших сред из периодических в непрерывные. Мы можем сделать это за счет создания конвейера развертывания, и он будет выполнять интеграцию нашего кода и сред и запускать серию тестов каждый раз, когда в систему контроля версий вносится новое изменение (рис. 13).



Рис. 13. Конвейер развертывания (источник: Humble and Farley, Continuous Delivery, 3)

Конвейер развертывания, впервые описанный Джезом Хамблом и Дэвидом Фарли в их книге *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, гарантирует, что весь код, записанный в систему контроля версий, автоматически собран и проверен в среде, близкой к производственной. Действуя таким образом, мы обнаруживаем любые ошибки сборки, тестирования или интеграции сразу же, как только они появились, что позволяет нам немедленно их исправить. При правильном выполнении процедур мы можем всегда быть уверенными, что код находится в состоянии готовности к развертыванию и релизу.

Для достижения этой цели необходимо создать автоматизированные процессы сборки и тестирования, выполняющиеся в выделенных средах. Это имеет жизненно важное значение по следующим причинам:

- процессы сборки и тестирования можно запустить в любое время и независимо от того, какой стиль работы предпочитают другие инженеры;
- раздельные процессы сборки и тестирования гарантируют, что мы понимаем все зависимости, необходимые для сборки, упаковки, запуска и тестирования нашего кода (то есть проблема «это работало на ноутбуке разработчика, но не работает в производственной среде» исключена);
- мы можем упаковать наши приложения, чтобы обеспечить повторяемость установки кода и конфигураций в разных средах (например, в Linux RPM, ум, прм, в Windows, OneGet, могут также использоваться альтернативные системы упаковки для интегрированных систем, такие как файлы EAR и WAR для Java, gems для Ruby и так далее);
- вместо того чтобы упаковывать наш код, мы можем помещать наши приложения в развертываемые контейнеры (например, Docker, Rkt, LXD, AMIs);
- среды могут быть сделаны более близкими к производственным, причем стабильным и повторяемым способом (например, из среды удаляются компиляторы, выключаются флаги отладки и тому подобное).

Конвейер развертывания после любого изменения проверяет, что код успешно интегрируется в среду, близкую к производственной. Он становится платформой. Через нее тестировщики запрашивают сборки и сертифицируют их во время приемочных испытаний и тестирования удобства использования, и он будет автоматически выполнять проверки производительности и безопасности.

Кроме того, он будет использоваться для самообслуживающихся сборок сред приемо-сдаточного тестирования, интеграционного тестирования и тестирования безопасности. На следующих шагах, по мере того как мы станем развивать конвейер развертывания, он также будет использоваться для управления всеми видами деятельности, необходимыми, чтобы провести сделанные изменения от системы контроля версий до развертывания.

Для обеспечения функциональности конвейера развертывания были разработаны различные продукты, в том числе с открытым исходным кодом (например, Jenkins, ThoughtWorks Go, Concourse, Bamboo, Microsoft Team Foundation Server, TeamCity, Gitlab CI, а также решения на базе облачных служб, таких как Travis CI and Snap).

Мы начнем создавать конвейер развертывания с той стадии записи изменений кода, когда делается сборка и создается установочный пакет, запускаются автоматизированные модульные тесты и выполняются дополнительные проверки, такие как статический анализ кода, анализ на дублирование кода, проверка тестового покрытия и проверка стилей. В случае успеха запускается стадия приемки: автоматически развертываются пакеты, созданные на этапе записи изменений, и запускаются автоматизированные приемочные тесты.

После того как изменения будут включены в систему контроля версий, желательно, чтобы упаковка кода делалась только один раз, с тем чтобы для развертывания кода по всей протяженности конвейера развертывания использовались те же самые пакеты. При этом код будет развертываться для интеграционного тестирования в тестовых средах точно так же, как и в производственной среде. Это уменьшает возможные отклонения, что позволяет избежать трудно диагностируемых ошибок на выходе процесса (например, с использованием разных компиляторов, флагов компиляции, разных версий библиотек или конфигураций).

Цель конвейера развертывания — предоставление каждому включенному в поток создания ценности, особенно разработчикам, как можно более быстрой обратной связи о том, что внесенные изменения нарушают готовность продукта к развертыванию. Это может быть изменение, внесенное в код, в любую из наших сред, в автоматизированные тесты или даже в инфраструктуру конвейера развертывания (например, настройки конфигурации Jenkins).

В результате инфраструктура нашего конвейера развертывания становится основой для процесса разработки как инфраструктура системы контроля версий. Конвейер развертывания также хранит историю каждой сборки, в том числе информацию, какие тесты были проведены с той или иной сборкой, какие сборки были развернуты и в какой среде, каковы результаты тестирования. Объединив это с информацией в истории контроля версий, мы можем быстро определить, что нарушило работу конвейера развертывания и, вероятно, как устраниТЬ ошибку.

Эта информация также помогает нам выполнить требования аудита и соответствия нормам и правилам, поскольку соответствующие подтверждения автоматически создаются в ходе повседневной работы.

Теперь, когда у нас есть работающая инфраструктура конвейера развертывания, мы должны создать методы непрерывной интеграции, обеспечивающие следующую функциональность:

- всеобъемлющий и надежный комплекс автоматизированных проверок, подтверждающий, что мы готовы к развертыванию;
- производственную культуру, «останавливающую всю производственную линию», когда тесты сообщают о сбое;
- разработчики трудятся над небольшими пакетами изменений основной ветви кода, а не над отдельной долгоживущей веткой для выделенного функционала.

В следующем разделе мы опишем, почему нам необходимо быстрое и надежное автоматизированное тестирование и как его создать.

На предыдущем шаге мы начали создавать инфраструктуру автоматизированного тестирования, проверяющую наличие зеленой сборки (такой, где все компоненты в системе контроля версий находятся в состоянии, обеспечивающем сборку и развертывание). Чтобы подчеркнуть, почему нам необходимо выполнять этот шаг — интеграции и тестирования — непрерывно, рассмотрим, что происходит, когда мы выполняем эту операцию лишь периодически, например в ходе ночных сборок.

Предположим, у нас есть команда из десяти разработчиков, и каждый ежедневно записывает изменения кода в системе контроля версий. Один вносит изменение, «ломающее» выполнение ночной сборки и работу по ее тестированию. В этом сценарии мы, прийдя утром на работу, обнаружим отсутствие «зеленой сборки», и потребуется несколько минут или, что более вероятно, несколько часов, пока команда выяснит, какие именно изменения вызвали проблему, кто их внес и как это исправить.

Рассмотрим наиболее плохой вариант, когда проблема не была вызвана изменением кода, а связана с изменением тестовой среды (например, какой-то параметр конфигурации был установлен неправильно). Команда разработчиков может считать, что они устранили проблему, поскольку все тесты успешно пройдены, но на следующий день обнаружится, что ночью вновь произошел сбой.

Можно еще сильнее усложнить ситуацию, предположив, что в течение дня записано десять изменений в коде. Каждое из них потенциально могло породить ошибки, нарушающие выполнение автоматизированных тестов, что еще увеличивает сложность успешной диагностики и устранения проблем.

Короче говоря, медленная и нечастая обратная связь убивает процесс разработки. Особенно в случае больших команд. Проблема еще усложняется, если десятки, сотни и даже тысячи разработчиков ежедневно записывают изменения кода в системе контроля версий. В результате

сборки и автоматические тесты часто сбоят, и разработчики даже приостанавливают на время запись изменений в системе контроля версий («зачем беспокоиться, ведь все равно сборка кода и его проверка сломаны»). Вместо этого для интеграции кода они дожидаются приближения конца проекта, что приводит ко всем нежелательным результатам крупных пакетов работы, интеграции в стиле big bang («большой взрыв») и проблемам с внедрением в производство.

Для предотвращения такого сценария нам нужны быстрые автоматические тесты, запускающиеся в средах сборки и тестирования всякий раз, когда в системе управления версиями записывается новое изменение. В этом случае мы можем найти и устранить любые проблемы немедленно, как это показывает пример команды GWS. Действуя так, можно обеспечить выполнение работы небольшими партиями, а также то, что в любой момент код готов к развертыванию.

В целом автоматизированные тесты относятся к одной из следующих категорий (перечислим, начиная от самых быстрых и до самых медленных).

- **Модульное тестирование (юнит-тесты).** Как правило, эти тесты проверяют один метод, класс или функцию изолированно от других, показывая разработчикам, что их код работает так, как задумано. По многим причинам, в том числе из-за необходимости поддерживать наши тесты быстрыми и не зависящими от состояния системы в целом, в модульных тестах используются заглушки для баз данных и других внешних зависимостей (например, функции изменены, чтобы возвращать статические, предустановленные значения вместо реального обращения к базам данных).

- **Приемочное тестирование.** Как правило, это тестирование приложения в целом. Оно необходимо, чтобы убедиться, что более верхнеуровневая функциональность работает так, как задумано (например, бизнес-критерии приемки в соответствии с требованиями клиента, правильность API), и что отсутствуют ошибки регрессии (то есть не повреждены функции, ранее работавшие правильно). Хамбл и Фарли так определили разницу между модульным и приемочным тестированием: «Цель модульного тестирования — показать, что отдельная часть приложения делает то, что задумал программист... Цель приемочных тестов — доказать, что наше приложение делает то, что от него ожидает клиент, а не то, что оно должно делать, по мнению программиста». После того как сборка проходит наши модульные тесты, конвейер развертывания запускает приемочное тестирование. Любая сборка, прошедшая приемочное тестирование, обычно становится затем доступной для тестирования вручную (например, исследовательское тестирование, тестирование пользовательского интерфейса и так далее), а также для интеграционного тестирования.

- **Интеграционное тестирование.** Интеграционное тестирование дает нам возможность убедиться, что наши приложения правильно взаимодействуют с другими приложениями и сервисами в производственной среде, в отличие от тестирования с заглушками на интерфейсах. Как отметили Хамбл и Фарли, «значительная часть работы в среде тестирования системной интеграции включает развертывание новых версий каждого приложения, пока они не начнут правильно взаимодействовать. В этой ситуации “смоук-тест” (проверка общей работоспособности) обычно — полный набор приемочных испытаний. Им подвергается все приложение». Интеграционным тестам подвергаются сборки, прошедшие модульные и приемочные испытания. Поскольку интеграционное тестирование часто оказывается нестабильным, мы хотим свести к минимуму количество интеграционных тестов и найти как можно больше дефектов в ходе модульного и приемочного тестирования. Возможность использования виртуальных или имитированных версий удаленных сервисов при запуске приемочных испытаний становится важным для архитектуры требованием.

Если разработчики сталкиваются с давлением из-за приближающегося срока завершения проекта, они могут перестать создавать модульные тесты в ходе повседневной работы, независимо от того, как мы определили состояние «сделано». Для обнаружения этой проблемы мы можем выбрать в качестве показателя и сделать прозрачной глубину покрытия тестами (как функцию от числа классов, строк кода, перестановок и так далее), даже считая наш набор приемосдаточных тестов не пройденным, если покрытие падает ниже определенного уровня.

Мартин Фаулер отмечает, что в целом «десять минут на сборку и тестирование — отличное время, в пределах разумного... Сначала мы выполняем компиляцию и запускаем более локальные модульные тесты с базой данных, полностью отключенной с помощью заглушек. Такие тесты могут выполняться очень быстро, завершаясь в течение десяти минут. Однако любые ошибки, связанные с более масштабным взаимодействием, особенно включающие работу с реальными базами данных, не будут найдены. На втором этапе сборка подвергается другому набору тестов приемочных испытаний, работающих с реальными базами данных и проверяющих более сложное, сквозное поведение. Выполнение этого набора тестов может занять несколько часов».

Конкретная цель разработки наших наборов автоматизированных тестов — найти ошибки на максимально раннем этапе тестирования. Вот почему мы запускаем быстро работающие автоматизированные тесты (например, модульные) раньше, чем медленнее работающие автоматизированные тесты (например, приемочные и интеграционные тесты), а они, в свою очередь, запускаются до любых видов тестирования вручную.

Другое следствие этого принципа — то, что любые ошибки должны быть найдены с помощью самых быстрых категорий тестирования. Если большинство ошибок обнаруживаются в ходе приемочного и интеграционного тестирования, то обратная связь к разработчикам приходит на порядок медленнее, чем при модульном тестировании — и интеграционное тестирование требует применения ограниченных и сложных тестовых сред. Они могут использоваться только одной командой в каждый момент, что еще сильнее затягивает получение обратной связи.

Кроме того, не только само воспроизведение ошибок, обнаруженных в ходе интеграционного тестирования, является трудоемким и отнимает много времени. Сложным является даже процесс проверки того, что они действительно исправлены (то есть разработчик создает исправление, но затем необходимо ждать четыре часа, чтобы узнать, успешно ли завершилось интеграционное тестирование).

Поэтому, обнаружив ошибку в ходе приемочного или интеграционного тестирования, мы должны создать модульный тест, чтобы он мог найти ошибку быстрее, раньше и дешевле. Мартин Фаулер описал понятие «пирамиды идеального тестирования». С ее помощью мы могли бы отлавливать большинство ошибок благодаря модульным тестам (рис. 14). На деле же зачастую верно обратное, и основной вклад в поиск ошибок вносят ручное и интеграционное тестирование.

Рис. 14. Пирамиды идеального и неидеального автоматизированного тестирования (источник: Martin Fowler, TestPyramid)

Если мы обнаружим, что модульные или приемочные испытания слишком сложны и дорогостоящи, чтобы писать их и поддерживать, то, скорее всего, у нас слишком связанная архитектура, когда четких границ между модулями не существует (или, возможно, никогда не существовало). В этом случае нам необходимо создать менее связанную систему. Ее модули можно тестировать независимо, без среды интеграции. Тогда можно сделать так, чтобы приемочные испытания даже самых сложных приложений выполнялись в течение нескольких минут.

Поскольку мы хотим, чтобы наши тесты выполнялись быстро, нам необходимо разработать их так, чтобы они могли работать параллельно и потенциально — на большом количестве разных серверов. Нам также может понадобиться выполнять тесты различных категорий параллельно. Например, когда сборка проходит приемочные тесты, мы можем запускать тесты производительности и одновременно — тесты безопасности, как показано на рис. 15. Мы можем допускать или не допускать исследовательское тестирование вручную до завершения всех автоматических проверок — если допускаем, это позволит раньше получить обратную связь, но может привести к затратам времени на сборки, не прошедшие автоматизированное тестирование.

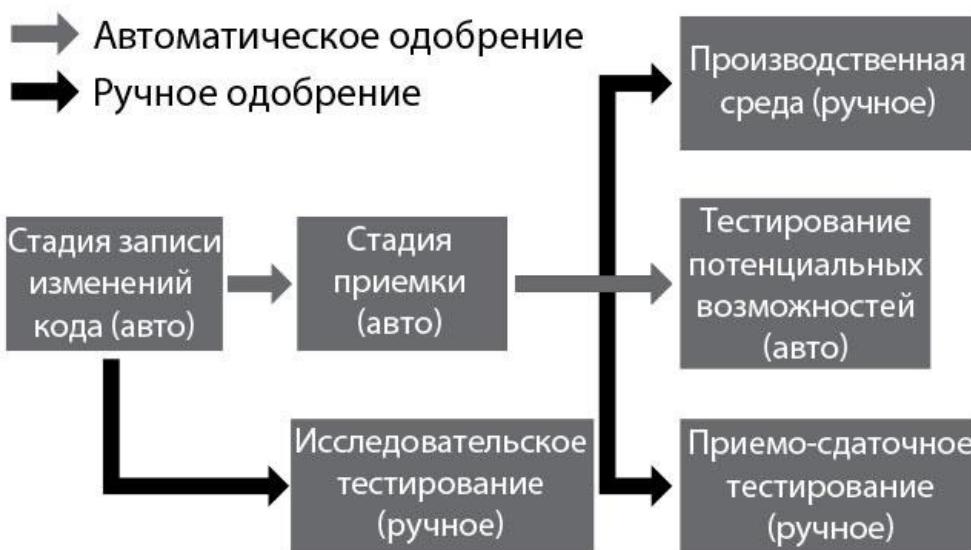


Рис. 15. Параллельный запуск автоматизированных тестов и тестирования вручную (источник: Джез Хамбл, Дэвид Фарли «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ»)

Любую сборку, прошедшую все наши автоматизированные тесты, мы делаем доступной для исследовательского тестирования, равно как и для других форм ресурсоемкого тестирования (вручную, например тестирование производительности). Мы хотим проводить такие тесты настолько часто, насколько возможно — либо непрерывно, либо по расписанию.

Любой, кто тестирует (включая и всех наших разработчиков), должен использовать самую последнюю сборку, прошедшую все автоматизированные тесты, а не ожидать, пока разработчики пометят конкретную сборку как готовую к тестированию. При этом мы можем обеспечить, чтобы процесс тестирования начался как можно раньше.

Один из наиболее эффективных путей обеспечения надежными автоматизированными тестами — написание тестов в ходе повседневной деятельности с использованием таких методов, как «разработка через тестирование» (TDD — test-driven development) и «разработка через приемочное тестирование» (ATDD — acceptance test-driven development). При использовании этих методов мы начинаем любое изменение в системе с того, что пишем автоматизированный тест, проверяющий, не будет ли сбоев в ожидаемом поведении кода, и лишь затем пишем код, который будет проходить эти тесты.

Этот метод был разработан Кентом Беком в конце 1990-х гг. как часть его концепции экстремального программирования и состоит из трех шагов.

1. Убедиться, что тест не пройден. «Напишите тест для проверки следующего кусочка функциональности, который вы хотите добавить». Запишите эти изменения.
2. Убедиться, что тест пройден. «Пишите функциональный код, пока тест не начнет успешно проходить». Запишите эти изменения.
3. Выполните рефакторинг как старого, так и нового кода, чтобы обеспечить его хорошую структурированность. Убедитесь, что тест успешно проходит. Снова запишите изменения кода.

Наборы автоматизированных тестов фиксируются в системе контроля версий наряду с нашим кодом, что обеспечивает документированность текущего состояния нашей системы. Разработчики, желающие понять, как использовать систему, могут обратиться к этим наборам тестов, чтобы найти рабочие примеры использования системных API.

Наша цель — найти как можно больше ошибок в коде с помощью наборов автоматизированных тестов, снижая зависимость от тестирования вручную. В своей презентации «В заботе о циклах обратной связи и их поддержании» (On the Care and Feeding of Feedback Cycles) на конференции Flowcon в 2013 г. Элизабет Хендриксон отмечала: «Хотя тестирование может быть автоматизировано, создание качества автоматизировать невозможно. Выполнение вручную тестов, нуждающихся в автоматизации, — пустая трата человеческого потенциала».

При этом мы даем возможность всем нашим тестирующим (разумеется, включая разработчиков) заниматься деятельностью, имеющей высокую ценность. Она не может быть автоматизирована: это аналитическое тестирование или улучшение самого процесса тестирования.

Однако простая автоматизация всех тестов, проводящихся вручную, может дать нежелательные результаты, ведь мы не хотим, чтобы автоматизированные тесты были ненадежны или давали ложно-положительный результат (то есть тесты должны быть проходимыми, только если код правильно функционирует, но должны сообщать о сбоях, если возникнут проблемы: низкая производительность, задержки при выполнении, неконтролируемое начальное состояние или непредусмотренное состояние из-за использования заглушек баз данных либо общих сред тестирования).

Ненадежные тесты, генерирующие ложные срабатывания, создают значительные проблемы — они отнимают драгоценное время (например, вынуждая разработчиков повторно запускать тест, чтобы определить, существует ли проблема на самом деле), увеличивают общее количество усилий, требующихся для запуска тестирования и интерпретации его результатов. Зачастую они же приводят к стрессовым нагрузкам на разработчиков; те начинают полностью игнорировать результаты тестов или выключают автоматическое тестирование и сосредоточиваются на создании кода.

Результат всегда один и тот же: мы обнаруживаем проблемы позднее, чем могли бы, их исправление оказывается более сложным делом, а наши заказчики получают неудачный результат, что, в свою очередь, создает излишнюю нагрузку на весь поток создания ценности.

Для смягчения ситуации предпочтительно иметь небольшое число надежных автоматизированных тестов, а не много проводимых вручную или ненадежных автоматических. Поэтому мы ориентированы на автоматизацию только тех тестов, которые действительно подтверждают желанные для нас бизнес-цели. Если отказаться от тестирования дефектов, обнаруживающихся в производственной среде, то мы должны добавить их обратно в набор тестов, осуществляемых вручную, и в идеале обеспечить их автоматизацию.

Гэри Грувер, ранее работавший вице-президентом по качеству разработок, релиза ПО и эксплуатации компании, так описывал свои впечатления: «На нашем сайте электронной коммерции крупного розничного продавца мы перешли от 1300 тестов, выполняемых вручную каждые десять дней, к десяти автоматизированным, запускаемым при каждой записи изменений кода. Гораздо лучше выполнить несколько надежных тестов, чем много ненадежных. С течением времени мы расширили этот набор до сотен тысяч автоматизированных тестов».

Другими словами, мы начинаем с небольшого числа надежных автоматических проверок и с течением времени увеличиваем их количество, все сильнее укрепляя уверенность, что мы быстро обнаружим любые изменения в системе, способные вывести ее из состояния готовности к развертыванию.

Слишком часто мы обнаруживаем во время интеграционного тестирования или уже после развертывания в производственную среду, что наше приложение имеет низкую производительность. Проблемы производительности зачастую трудно обнаружить, например, когда работа замедляется с течением времени, и они остаются незамеченными, пока не становится слишком поздно (к примеру, запросы к базе данных без использования индекса). И многие из этих проблем сложно решать, особенно когда они вызваны принятыми нами архитектурными решениями или непредвиденными ограничениями нашей сети, базы данных, системы хранения данных или других систем.

Наша цель — написать и запустить автоматические тесты производительности, проверяющие производительность всего стека приложения (код, базы данных, хранилища, сети, виртуализация и так далее) в рамках конвейера развертывания, чтобы мы могли обнаруживать проблемы на раннем этапе, когда внесение исправлений делается быстро и обходится малой ценой.

Поняв, как наше приложение и среды ведут себя под нагрузкой, близкой к реальной, мы можем гораздо лучше планировать мощности нашей системы, а также выявлять нижеперечисленные ситуации и подобные им:

- время выполнения запроса к базе данных растет нелинейно (например, мы забыли включить индексирование базы данных, и время загрузки страницы увеличивается с тридцати секунд до ста минут);
- изменение кода вызывает десятикратное увеличение количества вызовов базы данных, нагрузки на системы хранения или сетевого трафика.

Если у нас проводятся приемочные испытания и они могут выполняться параллельно, то мы используем их как основу для наших тестов производительности. Например, предположим, что мы работаем с сайтом электронной торговли и определили, что операции «поиск» и «оформить заказ» важны и должны хорошо выполняться даже под нагрузкой. Для проверки этого мы можем запустить одновременно тысячи приемочных тестов поиска и тысячи приемочных тестов оформления заказа.

Из-за большого объема вычислений и операций ввода-вывода, необходимых для выполнения тестов производительности, создание среды для такого тестирования может оказаться более сложным, чем создание производственной среды для самого приложения. Поэтому мы должны создавать среду для тестирования производительности в начале любого проекта и обеспечивать выделение всех ресурсов, необходимых, чтобы она функционировала корректно и на начальных этапах.

Чтобы в начале работы выявить проблемы с производительностью, мы должны регистрировать результаты тестов производительности и оценивать результаты каждого запуска по сравнению с предыдущими результатами. Например, мы можем посчитать, что тест не пройден, если производительность отличается более чем на 2 % от результатов предыдущего запуска.

В дополнение к тестированию того, что код работает, как запланировано, и выдерживает нагрузку, близкую к производственной, мы также хотели бы проверить все другие атрибуты системы, о которых считаем нужным позаботиться. Это так называемые нефункциональные требования: доступность, масштабируемость, производительность, безопасность и так далее.

Многие из этих требований оказываются выполненными при правильной конфигурации наших сред, поэтому мы должны также создавать автоматические проверки, чтобы убедиться, что наши среды были созданы и настроены правильно. Например, мы хотим обеспечить согласованность и правильность следующих характеристик (на их основе выполняются и многие нефункциональные требования, в частности безопасности, производительности и доступности):

- поддержка приложений и баз данных, библиотек и так далее;
- интерпретаторы языков программирования, компиляторы и так далее;
- операционные системы (например, включение ведения журналов аудита и так далее);
- все зависимости.

Используя инструменты автоматизированного управления конфигурацией «infrastructure as a code» (например, Puppet, Chef, Ansible, Salt, Bosh), мы можем задействовать те же инструменты тестирования, что и для проверки кода, чтобы также выяснить, что наши среды настроены и работают правильно (например, используя проверки конфигурации сред в тестах Cucumber или Gherkin).

Кроме того, подобно тому как мы используем средства анализа приложений в конвейере развертывания (например, статический анализ кода, анализ тестового покрытия), мы должны запускать инструменты, анализирующие код автоматизированной конфигурации (например, Foodcritic for Chef, Puppet-lint for Puppet). Мы должны также выполнить проверки усиления безопасности как часть наших автоматических тестов, чтобы убедиться, что все настроено надежно и правильно (например, конфигурации серверов).

В любой момент времени наши автоматизированные тесты могут подтвердить, что у нас есть «зеленая» сборка и она находится в готовности к развертыванию. Теперь мы должны создать шнур-андон, чтобы, когда кто-либо нарушил работу конвейера развертывания, мы смогли предпринять все необходимые шаги для возвращения обратно в «зеленое» состояние сборки.

Когда в конвейере развертывания «зеленая» сборка, мы обретаем высокую степень уверенности, что наши код и окружение при развертывании изменений в производственной среде будут работать именно так, как задумывалось.

Чтобы поддерживать конвейер развертывания в «зеленом» состоянии, создадим виртуальный шнур-андон, аналогичный физическому шнуру в системе производства Toyota. Когда кто-либо вносит изменение, нарушающее сборку или прохождение автоматизированных тестов, любая новая работа не подпускается в систему, пока проблема не устранена. И если кто-то нуждается в помощи для устранения этой проблемы, он может получить ее от всех членов команды, как и в примере с компанией Google, приведенном в .

Когда работа конвейера развертывания нарушена, мы по крайней мере должны уведомить о сбое всю команду, чтобы тот, из-за кого проблема возникла, мог ее исправить или откатить внесенные изменения. Мы даже можем настроить систему контроля версий для предотвращения дальнейшей записи изменений кода, пока первая стадия (то есть сборка и модульное тестирование) конвейера развертывания не перейдет обратно в «зеленое» состояние. Если эта проблема вызвана тем, что автоматизированная проверка выдала ложноположительную оценку, неправильная проверка должна быть переписана или удалена. Каждый член команды должен быть наделен полномочиями для совершения отката, чтобы вернуть сборку обратно в «зеленое» состояние.

Рэнди Шуп, бывший технический директор Google App Engine, писал о важности возвращения процесса развертывания обратно в «зеленое» состояние: «Мы ставим командные цели выше индивидуальных, когда мы можем помочь кому-нибудь из членов команды продвинуть его работу вперед, мы делаем это всей командой. Это правило применимо ко всем случаям, независимо от того, помогаем ли мы кому-то исправить сборку, автоматизированный тест или даже делаем для него обзор кода. И конечно же, мы уверены, что все будут делать то же самое для нас, если нам понадобится помочь. Эта система работала без особых формальностей или правил — все знали, что нашей задачей было не просто “написать код”, но “запустить сервис”. Вот почему мы сделали приоритетными все вопросы качества, особенно связанные с надежностью и масштабированием, и рассматривали их как наиболее приоритетные задачи, а их невыполнение — как ошибку, приводящую к неработоспособности системы. С точки зрения системы эти методы удерживали нас от соскальзывания назад».

Когда на более поздних этапах конвейера развертывания, таких как приемочные испытания или тесты производительности, происходит сбой, мы вместо остановки новых работ собираем

оперативно всех разработчиков и тестировщиков, несущих ответственность за немедленное устранение этих проблем. Они должны также создать новые тесты, выполняемые на более ранней стадии конвейера развертывания, чтобы отловить появление этих проблем при возможных регрессиях. Например, если мы обнаружим дефект в ходе приемочных испытаний, то должны написать модульный тест для раннего выявления проблемы. Аналогично, обнаружив дефект в ходе аналитического тестирования, мы должны написать модульный или приемочный тест.

Чтобы повысить видимость сбоев в ходе автоматизированного тестирования, создадим хорошо заметные индикаторы, чтобы вся группа могла увидеть, когда сборки или автоматические тесты дают сбой. Многие команды используют специальные лампы, расположенные на стене и указывающие текущий статус сборки. Есть и другие забавные способы уведомить команду, что сборка сломана: включить гелевые светильники, воспроизвести голосовую запись, песню или гудок автомобильного клаксона, использовать светофоры и так далее.

Во многих отношениях этот шаг более сложный, чем создание сборок и тестовых серверов — то были чисто технические мероприятия, а описываемый шаг требует изменения стимулов поведения членов команды. Вместе с тем непрерывная интеграция и непрерывная поставка требуют проведения этих изменений, и мы изучим причины такой необходимости в следующем разделе.

Если мы не потянем вовремя шнур-андон и, следовательно, не исправим немедленно какие-то проблемы конвейера развертывания, то столкнемся со слишком хорошо знакомыми проблемами — нам будет гораздо сложнее вернуть наши приложения и среды обратно в состояние готовности к развертыванию. Рассмотрим следующую ситуацию:

- кто-то записал изменения кода, «ломающие» сборку или автоматизированные тесты, но никто не исправляет ошибку;
- кто-то еще записывает другое изменение в «сломанной» сборке, также не проходящее автоматизированные тесты — но никто не видит результатов проверки этого кода, а ведь они дали бы возможность увидеть новый дефект, не говоря уже о его исправлении;
- имеющиеся у нас тесты не работают надежно, и поэтому маловероятно, что мы будем создавать новые тесты (зачем, если мы и имеющиеся-то тесты не можем заставить работать!).

Когда это происходит, развертывание в любой среде становится ненадежным, поскольку у нас нет автоматизированных тестов или мы использовали метод водопада, когда большинство проблем обнаруживаются уже в производственной среде. Неизбежный результат порочного цикла — то, что мы в конце концов оказываемся там, где начинали. На непредсказуемом «этапе стабилизации», занимающем недели или месяцы, вся наша команда оказывается в пучине кризиса, пытаясь обеспечить прохождение продуктом всех тестов, срезая острые углы под давлением приближающихся сроков и увеличивая размер нашего технического долга.

В этой главе мы рассмотрели создание всеобъемлющего набора автоматизированных тестов для подтверждения того, что у нас есть «зеленая» сборка, проходящая все тесты и находящаяся в состоянии, пригодном к развертыванию. Мы организовали выполнение этих тестов в рамках нашего конвейера развертывания. Мы также сделали нормой производственной культуры осуществление всего необходимого для возвращения сборки в «зеленое» состояние, если кто-то вносит изменение, «ломающее» любой из наших автоматизированных тестов.

Тем самым мы заложили основу для осуществления непрерывной интеграции, позволяющей большому количеству небольших команд самостоятельно и безопасно разрабатывать, тестировать и развертывать код в производственной среде, обеспечивая предоставление ценности клиентам.

## **Глава 11. Запустить и практиковать непрерывную интеграцию**

В предыдущей главе мы создали методы автоматизированного тестирования, чтобы обеспечить быстрое получение разработчиками обратной связи о качестве их труда. Это становится еще более важным по мере увеличения числа разработчиков и количества ветвей в системе контроля версий.

Возможность ветвления в системах контроля версий была создана в основном для обеспечения параллельной работы разработчиков над разными частями проекта без риска, что кто-то из них сможет внести изменения, дестабилизирующие основную ветку (иногда ее называют мастером или главной линией), или добавит в нее ошибки.

Однако чем дольше разработчикам разрешают изолированно работать над их ветками, тем труднее затем становится интегрировать и объединить в основную ветку проекта изменения, сделанные каждым. Фактически сложность интеграции растет экспоненциально по мере увеличения количества веток и количества изменений в каждой ветке.

Проблемы интеграции приводят к необходимости значительного количества переделок, чтобы привести продукт обратно в состояние пригодности к развертыванию, в том числе из-за противоречивых изменений (их приходится обновлять вручную), или изменений, слияние которых нарушает работу автоматизированных тестов или выполняемых вручную, и обычно переделки требуют усилий нескольких разработчиков. И поскольку интеграция традиционно выполняется в конце проекта, если она занимает намного больше времени, чем запланировано, то часто приходится чем-то жертвовать, чтобы выпустить продукт в срок.

Это приводит к другой нисходящей спирали: если слияние кода оказывается затруднено, мы стараемся выполнять его реже, что делает следующие слияния еще более трудными. Для решения этой проблемы была разработана непрерывная интеграция: включение включение разработческих веток в основную — часть повседневной работы.

Неожиданная широта проблем, решаемых с помощью непрерывной интеграции, а также сами решения подтверждены опытом Гэри Грювера, директора по проектированию компании HP LaserJet Firmware, создающей программное обеспечение для всех выпускаемых сканеров, принтеров и многофункциональных устройств.

Отдел состоял из четырех сотен разработчиков, находящихся в США, Бразилии и Индии. Несмотря на размер отдела, работа шла очень медленно. В течение ряда лет они были не в состоянии создать новые функциональные возможности так быстро, как это было необходимо для бизнеса.

Грювер так описывал проблему: «Маркетинг мог прийти к нам с миллионом идей о том, как поразить наших заказчиков, а мы отвечали: “Выберите из вашего списка две вещи, которые вы хотите получить в течение следующих шести — двенадцати месяцев”».

В год выпускалось всего две версии встроенного ПО, при этом основная часть времени затрачивалась на портирование кода для поддержки новых продуктов. По оценке Грювера, лишь 5 % общего времени тратилось на создание новых функциональных возможностей, а остальное время уходило на непродуктивные работы, связанные с погашением технического долга, такие как управление многочисленными ветками кода и тестирование вручную:

- 20 % на подробное планирование (низкая производительность и длительное время релиза приводили к ошибочным оценкам, и в надежде получить более устраивающий ответ сотрудникам отдела предлагали оценить работу более подробно);
- 25 % занимало портирование кода, поддерживаемое в отдельных ветках;
- 10 % уходило на интеграцию кода, написанного разработчиками в разных филиалах;
- 15 % занимало завершение тестирования вручную.

Грювер и его команда поставили цель увеличить долю времени, затрачиваемого на инновации и новые функциональности, в десять раз. Группа надеялась, что цель может быть достигнута посредством:

- непрерывной интеграции и разработкой на базе основной ветки;
- значительных инвестиций в автоматизацию тестирования;

- созданием эмуляторов оборудования, чтобы тесты можно было запускать на виртуальных платформах;
- воспроизведением сбоев в ходе тестирования на рабочих станциях разработчиков;
- разработкой новой архитектуры для поддержки работы всех принтеров на общих сборках и версиях.

Раньше релиз новой линейки продуктов требовал создания новой ветки кода, а каждая модель имела уникальное встроенное ПО. Его возможности определялись во время компиляции. Новая архитектура подразумевала, что все разработчики используют общую базу кода с единой версией встроенного программного обеспечения, поддерживающей все модели LaserJet, построенной из кода основной ветки, а особенности принтеров определяются во время выполнения из XML-файла конфигураций.

Четыре года спустя у них имелась одна база кода, поддерживающая все двадцать четыре продукта линеек HP LaserJet, разрабатываемая как в основной ветке. Грювер признавал, что разработка на основе основной ветки требует значительного изменения образа мыслей. Инженеры считали, что такой метод никогда не сможет работать, но после того, как начали его использовать, даже не помышляли вернуться к прежнему методу. За эти годы из HP уволились несколько инженеров, и они говорили, насколько устарела организация разработки в новых компаниях, как трудно быть эффективным и выпускать хороший код при отсутствии обратной связи, предоставляемой непрерывной интеграцией.

Однако разработка на базе основной ветки потребовала создания более эффективного автоматизированного тестирования. Грювер отмечал: «Без автоматического тестирования и непрерывной интеграции это оказывается самым быстрым способом получить большую кучу барахла. Оно никогда не скомпилируется или не будет работать правильно». Вначале цикл полного тестирования вручную требовал шести недель.

Чтобы иметь возможность автоматически протестировать все сборки микропрограмм, значительные средства вложили в эмуляторы принтеров и за шесть недель создали тестовые фермы — в течение нескольких лет две тысячи эмуляторов принтеров запускались на шести стойках с серверами и обрабатывали сборки микропрограмм, взятых из конвейера разработки. Их система непрерывной интеграции (CI) выполняла весь комплекс автоматизированного модульного, приемочного и интеграционного тестирования сборок, сделанных на базе основной ветки, как описано в предыдущей главе. Кроме того, создали культуру производства, останавливающую всю работу, как только кто-то из разработчиков повреждал конвейер развертывания. Так обеспечивался быстрый возврат системы в рабочее состояние.

Автоматизированное тестирование создает быструю обратную связь, позволяющую разработчикам быстро убедиться, что зафиксированный код действительно работает. Модульное тестирование может выполнятся на рабочих станциях за минуты, три уровня такого тестирования можно запускать после каждой фиксации кода или каждые два-четыре часа. Завершающий полный регрессионный тест можно запускать каждые двадцать четыре часа. В ходе этого процесса разработчики:

- сокращают количество сборок до одной в день, что в конечном счете означает от десяти до пятнадцати сборок в день;
- переходят от примерно двадцати фиксаций кода в день, совершаемых «главным по сборкам», до свыше ста в день, совершаемых самими разработчиками;
- получают возможность изменять или добавлять до 75–100 тысяч строк кода каждый день;
- снижают длительность регрессионного тестирования с шести недель до одного дня.

Этот уровень производительности невозможно обеспечить до внедрения непрерывной интеграции, когда одно только создание работающей сборки требовало нескольких дней упорного труда. В результате полученные бизнесом преимущества удивляют:

- доля времени разработчиков, затраченного на инновации и написание новых функциональностей, возросла с 5 до 40 %;

- общая стоимость разработки была сокращена примерно на 40 %;
- количество программ, находящихся в разработке, увеличилось примерно на 140 %;
- расходы на разработку одной программы сократились на 78 %.

Опыт, полученный Грювером, показывает: после полномасштабного использования системы контроля версий непрерывная интеграция — один из наиболее важных методов, обеспечивающих быстрый поток работы в нашем потоке создания ценности, позволяя многим командам разработчиков самостоятельно разрабатывать, тестировать и доставлять продукт. Тем не менее непрерывная интеграция остается противоречивым методом. В оставшейся части главы описываются приемы, необходимые для внедрения непрерывной интеграции, а также то, как преодолеть обычно возникающие возражения.

Как описано в предыдущих главах, когда в систему контроля версий вносятся изменения, нарушающие работу конвейера развертывания, мы быстро набрасываемся на проблему, чтобы ее исправить, в результате чего конвейер развертывания возвращается в «зеленое» состояние. Вместе с тем значительные проблемы появляются, если разработчики трудятся над долго существующими частными ветками (их еще называют «ветки функциональности»). Их обратное слияние с основной веткой выполняется нерегулярно, в результате чего список изменений оказывается весьма велик. Как показано на примере HP LaserJet, это приводит к хаосу и значительным переделкам, требующимся для приведения кода в состояние готовности к релизу.

Джефф Этвуд, основатель сайта Stack Overflow («Переполнение стека») и автор блога Coding Horror («Ужас кодирования»), отмечает, что, в то время как существует множество стратегий ветвления, все они могут быть поделены на две группы.

- **Оптимизация для индивидуальной производительности.** Каждый участник проекта работает над своей собственной личной веткой. Все работают независимо, и никто не может повредить чужую работу, однако слияние веток становится кошмаром. Совместная работа становится крайне трудной — чтобы увидеть даже мельчайшую часть полной системы, надо работу каждого человека аккуратно объединить с работой остальных.
- **Оптимизация для производительности команды.** Все члены команды работают в одной и той же общей области. Никаких ветвлений нет, есть длинная, не разделенная на части прямая линия разработки. Не надо ни в чем разбираться, поэтому фиксация кода проста, но любая фиксация может сломать весь проект и резко затормозить весь прогресс команды.

Мнение Этвуда абсолютно правильно, и если сформулировать точнее, то усилия, необходимые для успешного объединения веток, растут экспоненциально по мере роста числа веток. Проблема заключается не только в объеме переделок, необходимых после «ада слияний», но также в том, что обратную связь от конвейера развертывания мы получаем с задержкой. Например, скорее всего, только в конце процесса разработки удастся выполнить тестирование производительности полностью интегрированной системы — непрерывным данный процесс сделать не получится.

Кроме того, если мы повышаем скорость создания кода, увеличивая количество разработчиков, то усиливаем вероятность того, что любое изменение затронет какого-то другого разработчика и увеличит количество сбоев в конвейере развертывания.

Это один из побочных эффектов слияния больших наборов изменений, вызывающих наибольшие сложности: когда слияние трудно, у нас остается меньше возможностей и, кроме того, понижается мотивация улучшать код и выполнять его рефакторинг, поскольку рефакторинг, скорее всего, будет означать, что кому-то придется переделывать работу. Когда такое происходит, желание модифицировать код, имеющий зависимости во всем репозитории кода, заметно ослабевает, хотя (и в этом трагизм ситуации) именно это могло бы дать нам наибольшую отдачу.

Вот как Уорд Каннингем, разработчик первой wiki, впервые описал технический долг: если мы не выполняем агрессивный рефакторинг нашей базы кода, то с течением времени становится все труднее вносить изменения в код и поддерживать его, а это снижает скорость добавления новых функциональных возможностей. Решение этой проблемы — одна из основных причин создания непрерывной интеграции и практики развития кода на базе основной ветки и оптимизации производительности команды вместо индивидуальной производительности.

Контрмеры по отношению к большим пакетам слияний — это организация непрерывной интеграции и методы разработки на базе основной ветки, когда все разработчики фиксируют свой код на

основной ветке по меньшей мере один раз в день. Частая фиксация кода значительно уменьшает размер одного пакета, и команда разработчиков оказывается способна выполнить задание за день. Чем чаще разработчики фиксируют код в основной ветке, тем меньше размер пакета и тем ближе мы к теоретическому идеалу штучного потока.

Частая фиксация кода в основной ветке означает: мы можем запускать все автоматические проверки системы программного обеспечения как целого и получать уведомления, если изменение «ломает» работу других частей приложения или вмешивается в работу другого разработчика. И поскольку можно обнаруживать проблемы слияния, пока они невелики, мы способны исправить их быстрее.

Мы даже можем настроить конвейер развертывания так, чтобы он отвергал любые фиксации (например, кода или изменений среды), нарушающие состояние готовности к развертыванию. Этот способ называется *управляемой фиксацией* (gated commits): конвейер развертывания сначала проверяет, будут ли все переданные изменения успешно объединены, пройдет ли сборка так, как ожидалось, и выполнится ли автоматическое тестирование, прежде чем изменения будут внесены в основную ветку. Если на одном из этапов проверка не пройдет, то разработчик получит уведомление и сможет внести исправления, не затрагивая никаких других элементов потока создания ценности.

Принятая практика ежедневной фиксации кода также заставляет нас разделять работу на небольшие кусочки, в то же время обеспечивая нахождение основной ветки в рабочем состоянии, годном к релизу.

И система контроля версий становится неотъемлемым механизмом обмена информацией внутри команды — каждый ее член лучше понимает систему в целом, находится в курсе состояния конвейера развертывания, и они могут помочь друг другу, когда он ломается. В результате получаем более высокое качество и небольшое время разработки.

Рассмотрев эти методы, мы можем теперь снова изменить определение состояния «сделано» (выделено жирным шрифтом): «В конце каждого интервала разработки мы должны иметь интегрированный, протестированный, рабочий и потенциально готовый к поставке (что демонстрируется запуском в среде, близкой к производственной) код, **созданный на базе основной ветки процессом, “запускаемым одним щелчком мыши”, и проверенный автоматическими тестами**».

Пересмотренное определение поможет нам в дальнейшем гарантировать, что написанный нами код всегда пройдет тестирование и будет пригоден к развертыванию. Поддерживая код в готовности к развертыванию, мы способны исключить традиционные методы, когда тестирование кода и его стабилизация осуществляются отдельными фазами в конце проекта.

Эрнест Мюллер, помогавший производить DevOps-трансформацию в компании National Instruments, позже, в 2012 г., занимался преобразованием процессов разработки и релиза ПО в компании Bazaarvoice. Она обеспечивала обработку пользовательского контента (например, обзоры, рейтинги) для тысяч предприятий розничной торговли, в том числе Best Buy, Nike и Walmart.

В то время Bazaarvoice имела 120 миллионов долларов дохода и готовилась к IPO. Основным источником прибыли компании было приложение Bazaarvoice Conversations — монолитное приложение на языке Java, состоявшее из почти пяти миллионов строк кода, написанных начиная с 2006 г. и содержащихся в 15 тысячах файлов. Оно работало на 1200 серверах в четырех центрах обработки данных с использованием нескольких поставщиков облачных услуг.

В компании, частично в результате перехода на Agile-процесс и двухнедельный цикл разработки, появилось сильное стремление увеличить частоту релизов по сравнению с действовавшим планом релиза версий каждые десять недель. Сотрудники также приступили к разделению частей монолитного приложения, разбивая его на микросервисы.

В первый раз они попытались перейти на двухнедельный график в январе 2012 г. Мюллер отмечал: «Дело не заладилось. Попытка вызвала массовый хаос, мы получили от наших клиентов сорок четыре сообщения о неполадках. Лейтмотивом реакции менеджеров было “давайте не будем повторять такие попытки”».

Мюллер стал руководить процессом релиза. Он поставил целью делать релизы каждые две недели, не допуская простоев при обслуживании клиентов. В бизнес-цели увеличения частоты релизов входило использование более быстрого A/B-тестирования (описано в следующих главах) и увеличение скорости выпуска нового функционала. Мюллер определил три основные проблемы:

- недостаточная автоматизация тестирования на любом уровне тестирования в течение двухнедельного интервала не дает возможности предотвратить крупномасштабные сбои;

- стратегия ветвлений системы контроля версий позволяет разработчикам проверить новый код только почти непосредственно перед релизом;
- команды, работающие над микросервисами, делали релизы независимо, что нередко вызывало проблемы с монолитной версией, и наоборот.

Мюллер пришел к выводу: процесс развертывания монолитного приложения *Conversations* необходимо стабилизировать, что требует введения непрерывной интеграции. В течение следующих шести недель разработчики прекратили добавление новых функциональных возможностей, вместо этого сосредоточившись на написании наборов автоматизированного тестирования, включая модульное тестирование в JUnit, регрессивное тестирование в Selenium, и запустив работу процесса развертывания в TeamCity. «Все время выполняя эти тесты, мы чувствовали, что можем вносить изменения относительно безопасно. И самое главное, мы смогли бы сразу же обнаружить, когда нечто повреждало что-то, тогда как раньше обнаруживали это только после запуска в производство».

Они также перешли на релиз кода на базе основной ветки. Каждые две недели создавали новую специальную ветку, но затем в ней не разрешалось, кроме самых крайних случаев, делать никаких новых изменений кода: все изменения обрабатывались при выходе из процесса, или по заданию на работу, или внутри команд через их внутренние wiki. Эта ветка проходила тестирование, а затем передавалась в производство.

Увеличение предсказуемости и улучшение качества релизов оказались поразительными:

- релиз в январе 2012 г.: 44 обращения о сбоях от клиентов (начаты усилия по внедрению постоянной интеграции);
- релиз 6 марта 2012 г.: спустя пять дней — пять обращений о сбоях от клиентов;
- релиз 22 марта 2012 г.: сделан вовремя, одно обращение от клиента;
- релиз 5 апреля 2012 г.: сделан вовремя, обращений от клиентов нет.

Позже Мюллер описывал, насколько успешными оказались усилия:

«Мы добились такого успеха с релизами каждые две недели, что решили перейти на еженедельные релизы, что не требовало почти никаких изменений в инженерных командах. Поскольку релизы стали рутинным делом, было очень просто удвоить их количество в календарном плане и делать их тогда, когда об этом напоминал календарь. Действительно, это происходило практически без каких-то значимых событий. Большинство необходимых изменений делались в командах клиентского обслуживания и маркетинга. Они должны были вносить изменения в свои процессы, такие как изменение расписания еженедельных сообщений клиентам, какие изменения в функциях будут произведены. После этого мы начали работать над достижением следующих целей, в итоге приведших к уменьшению времени на тестирование с трех с лишним часов до менее часа, сокращению количества сред с четырех до трех (среда для разработки, для тестирования, производственная среда, а среда для завершающего тестирования была исключена) и переходу к полной модели непрерывной доставки, чтобы мы могли выполнить развертывание быстро, одним щелчком мыши».

Разработка на базе основной ветки — наиболее спорный метод из описанных. Многие инженеры не считают это возможным, среди них есть даже те, кто предпочитает работать над своей веткой, не отвлекаясь на другие и не имея дел с другими разработчиками. Однако данные из доклада 2015 State of DevOps Report компании Puppet Labs ясно показывают: разработка на базе основной ветки предсказывает более высокую производительность, лучшую стабильность и даже более высокую удовлетворенность разработчиков своей работой и меньшую вероятность появления у них эмоционального истощения.

Хотя вначале может оказаться трудным убедить разработчиков, рано или поздно они увидят исключительные преимущества такого подхода и, скорее всего, станут сторонниками постоянных преобразований, как это показывают примеры HP LaserJet и Bazaarvoice. Методы непрерывной интеграции создают основание для следующего шага — автоматизации процесса развертывания и низкого риска при релизе ПО.



## Глава 12. Автоматизация и запуск релизов с низким уровнем риска

Чак Росси — технический директор в Facebook. Одна из его задач — контроль за ежедневным релизом кода. В 2012 г. Росси так описал процесс: «Начиная с часа дня я переключаюсь в “режим эксплуатации” и вместе с командой готовлюсь запустить обновления, которые выйдут на в этот день. Эта наиболее напряженная часть работы в значительной степени зависит от решений моей команды и от прошлого опыта. Мы стремимся убедиться, что каждый, кто вносил релизные изменения, отчитывается за них, а также активно тестирует и поддерживает изменения».

Незадолго до релиза все разработчики релизных изменений должны появиться на своем канале интернет-чата — отсутствующим разработчикам приходит возврат обновлений, автоматически удаленных из пакета развертывания. Росси продолжает: «Если все хорошо и наши инструменты тестирования и “канареечные тесты” успешны, мы нажимаем большую красную кнопку, и весь “парк” серверов получает новый код. В течение 20 минут тысячи и тысячи компьютеров переходят на новый код без какого-либо заметного влияния на тех, кто использует сайт».

Позже в том же году Росси увеличил частоту релизов программного обеспечения до двух раз в день. Он объяснил, что второй релиз кода позволил инженерам, находящимся не на Западном побережье США, «вносить изменения и отправлять их так же быстро, как любой другой инженер в компании», а также предоставил всем еще одну возможность каждый день отправлять код и запускать новые функциональные возможности.

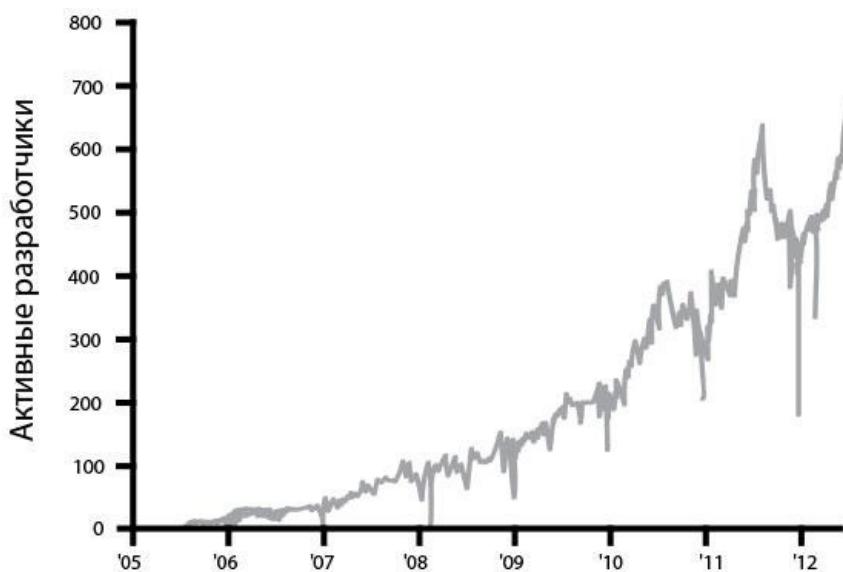


Рис. 16. Количество разработчиков Facebook в неделю, развертывающих свой код (источник: Chuck Rossi, Ship early and ship twice as often)

Кент Бек, создатель методологии экстремального программирования, один из основных сторонников разработки на основе тестирования и технический наставник в компании Facebook, прокомментировал стратегию релиза кода компании в статье, опубликованной на его странице в Facebook: «Чак Росси отметил: создается впечатление, что за одно развертывание Facebook может обработать только ограниченное количество изменений. Если мы хотим сделать больше изменений, то нам нужно выполнить больше развертываний. Это привело к неуклонному росту темпов развертывания в течение последних пяти лет, с одного раза в неделю до ежедневного, а затем — до трех раз в день для кода PHP и с шестинедельного до четырехнедельного, а затем двухнедельного цикла развертывания мобильных приложений. Улучшение было обеспечено главным образом инженерной группой, ответственной за релизы».

Используя непрерывную интеграцию и развертывание кода с помощью процесса с низким уровнем риска, компания Facebook сделала развертывание кода частью повседневной работы разработчиков, чем способствовала их производительности. Для этого необходимо, чтобы развертывание кода было автоматизированным, повторяемым и предсказуемым. Хотя в методиках, описанных выше, наши код и среды тестировались вместе, скорее всего, развертывание в производство выполнялось не очень часто, поскольку делалось вручную, занимало много времени, было трудным и утомительным, оказывалось подвержено ошибкам и часто вытекало из неудобной и ненадежной передачи работы от отдела разработки к отделу эксплуатации.

И поскольку все это непросто, мы склонны делать так все реже, что приводит к другой самоусиливающейся нисходящей спирали. Откладывая развертывание изменений в производство,

мы накапливаем все больше различий между кодом, который будет развернут, и кодом, в данный момент работающим в производстве, увеличивающим размер пакетной работы. Так растет риск неожиданных результатов, вызванных изменением, а также трудности, связанные с их исправлением.

В этой главе мы снизим внутренние трения, связанные с развертыванием в производство, чтобы оно могло выполняться часто и легко, как разработчиками, так и отделом эксплуатации. Мы можем сделать это за счет расширения конвейера развертывания.

Вместо простой непрерывной интеграции кода в среде, близкой к производственной, обеспечим продвижение в производство любой сборки, проходящей автоматизированные тесты и процессы валидации или по запросу (например, нажатием на кнопку), или автоматически (например, любая сборка, которая проходит все тесты, развертывается автоматически).

Поскольку в данной главе описано значительное количество методик, то, чтобы не прерывать представления концепций, будут использоваться обширные сноски, многочисленные примеры и дополнительная информация.

Достижение таких же результатов, как в компании Facebook, требует наличия автоматизированного механизма, развертывающего код в производственную среду. Если процесс развертывания существует много лет, то особенно необходимо в полной мере документировать все его этапы, например в картах процесса создания продукта. Их мы можем собрать в ходе рабочих совещаний или постепенно пополняя документацию (например, в форме wiki).

После того как мы задокументировали процесс, наша цель — упростить и автоматизировать как можно больше выполняемых вручную шагов, а именно:

- упаковка кода подходящим для развертывания образом;
- создание предварительно настроенных образов виртуальных машин или контейнеров;
- автоматизация развертывания и настройки конфигурации промежуточного ПО;
- копирование пакетов или файлов на производственные серверы;
- перезапуск серверов, приложений или служб;
- создание файлов конфигурации из шаблонов;
- запуск автоматизированных smoke-тестов, чтобы убедиться, что система работает и правильно настроена;
- запуск процедур тестирования;
- сценарии и автоматизация миграции базы данных.

Где это возможно, будем перерабатывать архитектуру с целью удаления некоторых шагов, в частности требующих длительного времени. Желая уменьшить опасность ошибок и потери знаний, мы также хотели бы сократить не только сроки выполнения, но и в максимально возможной степени — количество случаев, когда работа передается от одного отдела другому.

Если разработчики сосредоточатся на автоматизации и оптимизации процесса развертывания, это может привести к существенным улучшениям потока развертывания, таким как отсутствие необходимости новых развертываний или создание новых сред при небольших изменениях конфигурации приложений.

Однако для этого необходимо, чтобы разработчики действовали в тесном контакте с отделом эксплуатации для обеспечения того, чтобы все инструменты и процессы, созданные нами, могли использоваться в ходе процесса и чтобы отдел эксплуатации не отчуждался от общего процесса и не надо было заново изобретать колесо.

Многие инструменты, обеспечивающие непрерывную интеграцию и тестирование, также поддерживают возможность расширить конвейер развертывания, так что проверенные сборки могут быть введены в производство, обычно после приемочных испытаний (например, Jenkins Build Pipeline plugin, ThoughtWorks Go.cd and Snap CI, Microsoft Visual Studio Team Services и Pivotal Concourse).

Среди требований к нашему конвейеру развертывания следующие:

- **развертывание одинаковым образом в любой среде:** с помощью одного и того же механизма развертывания для всех сред (например, разработки, тестирования и производственной) наше развертывание в производство может быть гораздо более успешным, поскольку мы знаем, что оно уже было успешно выполнено не один раз в начале конвейера;
- **smoke-тестирование наших развертываний:** в ходе процесса развертывания мы должны протестировать, что можем подключаться к любой системе поддержки (например, базам данных, шинам сообщений, внешним сервисам), и запустить одну тестовую транзакцию через систему, чтобы убедиться, что наша система работает так, как задумано. Если любой из этих тестов не пройден, то мы должны остановить развертывание;
- **поддержание согласованных сред:** на предыдущих шагах мы создали процесс сборки среды за один шаг, с тем чтобы среды разработки, тестирования и производственные имели общий механизм их создания. Мы должны постоянно обеспечивать синхронизированность сред.

Конечно, в случае возникновения каких-либо проблем в ходе развертывания мы потянем шнур-андон и дружно набросимся на проблему, пока она не будет решена. Так же мы поступаем, если конвейер развертывания сбоит на любом из предыдущих шагов.

Компания CSG International — один из крупнейших операторов по печати счетов в США. Скотт Праф, главный архитектор и вице-президент отдела разработки ПО, в рамках усилий по повышению предсказуемости и надежности выпусков программного обеспечения удвоил частоту релизов с двух до четырех в год (уменьшив интервал между развертываниями вдвое, с 28 недель до 14).

Хотя команды разработчиков использовали непрерывную интеграцию для ежедневного развертывания их кода в средах тестирования, релизы в производственную среду выполнялись отделом эксплуатации. Праф отмечал: «Это как если бы у нас была "учебная команда", ежедневно (или даже чаще) тренировавшаяся создавать низкорисковые тестовые среды, совершенствуя свои процессы и инструменты. Но производственная "играющая команда" упражнялась очень редко, всего два раза в год. Что еще хуже, она тренировалась в высокорисковой производственной среде, часто сильно отличавшейся от предпроизводственных сред, имевших различные ограничения — в средах разработки отсутствовали многие производственные ресурсы (средства обеспечения безопасности, брандмауэры, средства балансировки нагрузки и SAN)».

Чтобы решить эту проблему, была создана общая команда эксплуатации (Shared Operations Team, SOT), отвечавшая за управление всеми средами (разработки, тестирования, производственной), ежедневно выполняя развертывание в средах разработки и тестирования, а каждые четырнадцать недель — развертывание в производство. Поскольку SOT выполняла развертывание каждый день, любые проблемы, с которыми она сталкивалась и которые не были исправлены, на следующий день возникали снова. Это побудило к автоматизации трудоемких или чреватых ошибками операций, проведенных вручную, и устранению любых проблем, которые потенциально могли бы повториться. Поскольку развертывание выполнялось почти сто раз перед релизом, большинство проблем задолго до него были найдены и устраниены.

Это позволило вытащить на свет проблемы, ранее известные только отделу эксплуатации. Но они тормозили весь поток создания ценности, и их надо было решать. Ежедневное развертывание обеспечивает ежедневную же обратную связь о том, какие методы сработали, а какие нет.

Исполнители также сосредоточили внимание на том, чтобы сделать все среды как можно более аналогичными и как можно скорее, включая ограниченные права доступа и средства балансировки нагрузки. Праф пишет: «Мы настолько приблизили непроизводственные среды к производственной, насколько смогли, и стремились как можно точнее воспроизвести в них ограничения, имеющиеся в производственной среде. Ранее использование сред, сходных с производственными, вызвало изменение архитектурных решений с целью сделать их более приемлемыми в ограниченных или отличающихся средах. При таком подходе каждый человек становится умнее».

Праф также отмечает:

«Мы столкнулись со многими случаями, когда изменялись схемы баз данных — либо 1) переданные в группу ДБА запросы в стиле "идите и разбирайтесь", либо 2) автоматизированные тесты, работавшие с неоправданно малыми наборами данных (например, сотни мегабайт вместо сотен гигабайт), что вело к сбоям в производственном процессе. При прежнем способе работы это приводило к дискуссиям между командами до глубокой ночи в поисках козла отпущения и попытках размотать клубок проблем. Мы создали процесс разработки и развертывания, снявший необходимость передачи группе ДБА заданий на работу благодаря перекрестному обучению разработчиков, автоматизации изменения схемы и ежедневному выполнению этого процесса. Мы

создали реалистичное тестирование нагрузки, не урезая данные заказчика, и в соответствии с идеальным решением выполняли миграцию каждый день. Благодаря этому мы сотни раз запускаем сервис с реалистичными сценариями еще до того, как он начнет обслуживать реальный производственный трафик».

Полученные результаты были удивительными. Ежедневно выполняя развертывание и удвоив частоту релизов в производственную среду, удалось снизить количество сбоев на производстве на 91 %, MTTR уменьшилось на 80 %, а время развертывания служб в производство уменьшилось с четырнадцати дней до одного и проходит в режиме полного исключения операций, проводимых вручную.

Праф сообщил: развертывание стало настолько рутинным процессом, что в конце первого дня команда эксплуатации играла в видеоигры. В дополнение к тому, что развертывание стало проходить более гладко и для разработчиков, и для эксплуатации, в 50 % случаев заказчик получал свою ценность за время, вдвое меньшее, чем обычно, что подчеркивало, насколько более частое развертывание полезнее для разработчиков, тестировщиков, отдела эксплуатации и клиентов.



Рис. 17. Ежедневные развертывания и увеличение частоты релизов уменьшают число инцидентов в производстве и MTTR (источник: DOES15 — Scott Prugh & Erica Morrison — Conway & Taylor Meet the Strangler (v2.0), видео на YouTube, 29:39, размещено DevOps Enterprise Summit, 5 ноября 2015 г.)

Рассмотрим утверждение Тима Тишлера, директора по автоматизации эксплуатации компании Nike, об общем опыте поколения разработчиков: «Я никогда не испытывал большего удовлетворения своей деятельностью как разработчика, чем когда писал код, нажимал кнопку для развертывания и мог видеть производственные показатели, подтверждающие, что код действительно работает в производственной среде, и чем когда сам мог все исправить, если он все-таки не работал».

Возможности разработчиков самостоятельно развернуть код в производство, быстро увидеть, довольны ли клиенты новыми функциями, и быстро устраниТЬ любые проблемы без необходимости создавать задание для отдела эксплуатации уменьшились в течение последнего десятилетия — отчасти в результате контроля и надзора, возможно, вызванных нормами безопасности, и необходимостью соответствовать требованиям.

В результате общая практика — развертывание кода отделом эксплуатации, из-за разделения обязанностей — широко признанная практика в целях снижения риска сбоев производства и обманов. Однако для достижения результатов DevOps наша цель — начать доверять другим механизмам управления, которые могут смягчать риски в той же степени или даже более эффективно, например посредством автоматизированного тестирования, автоматического развертывания и экспертной оценки изменений. Доклад 2013 State of DevOps Report, подготовленный компанией Puppet Labs по результатам опроса более четырех тысяч профессионалов в области технологий, показал, что нет статистически значимой разницы в изменении показателей успешности между теми организациями, где развертывание производится отделом эксплуатации, и теми, где развертывание выполняют разработчики.

Другими словами, при наличии общих целей у разработчиков и отдела эксплуатации и тогда, когда существует прозрачность, ответственность и возможность учета результатов развертывания, уже не важно, кто именно выполняет развертывание. Фактически даже те, кто выполняет другие роли, —

тестировщики и менеджеры проектов — способны развертывать определенные среды, чтобы иметь возможность быстро завершить работу, например настройку демонстрации конкретных функций в тестах или средах приемочного тестирования.

Чтобы создать лучшие условия для быстрого потока работы, нам нужен процесс продвижения кода по этапам. Он может быть выполнен и разработчиками, и отделом эксплуатации, причем в идеале без каких-либо действий вручную или передачи работы от одного сотрудника другому. Это влияет на следующие шаги:

- **сборка**: наш конвейер развертывания должен создавать пакеты из кода, взятого в системе контроля версий. Они могут быть развернуты в любой среде, включая производственную;
- **тестирование**: любой работник должен иметь возможность запускать любой или все наши наборы автоматизированных тестов на своей рабочей станции или на наших тестовых системах;
- **развертывание**: любой работник должен иметь возможность быстро развертывать эти пакеты в любой среде, к которой он имеет доступ, с помощью сценариев, также зафиксированных в системе контроля версий.

Эти методы позволяют успешно выполнить развертывание независимо от того, кто его будет выполнять.

После того как процесс развертывания кода автоматизирован, мы можем сделать его частью конвейера развертывания. В связи с этим наша автоматизация развертывания должна обеспечивать следующие возможности:

- убедиться, что пакеты, созданные во время непрерывного процесса интеграции, подходят для развертывания в производство;
- с одного взгляда показать готовность производственных сред;
- обеспечить запускаемый одним нажатием кнопки и самообслуживаемый метод развертывания в производство любой подходящей версии упакованного кода;
- автоматически записывать для аудита и обеспечения соответствия требованиям, какие команды были выполнены и на какой машине, когда, кто санкционировал и каков был результат;
- выполнить smoke-тест, чтобы убедиться, что система работает правильно и параметры конфигурации, в том числе такие элементы, как строки подключений к базам данных, корректны;
- обеспечить быструю обратную связь для работника, чтобы он мог быстро определить, осуществилось ли развертывание (например, успешно ли оно прошло, работает ли приложение в производственной среде в соответствии с ожиданиями и так далее).

Цель — обеспечить, чтобы развертывание выполнялось быстро, мы не хотим ждать часами, пока сможем определить, успешно ли развертывание нашего кода или нет, а затем еще столько же времени тратить на развертывание кода, исправляющего обнаруженные ошибки. Сейчас мы имеем такие технологии, как контейнеры, и можем завершить развертывание даже в самых сложных средах в течение нескольких секунд или минут. Данные, опубликованные в докладе 2014 State of DevOps Report, подготовленном компанией Puppet Labs, свидетельствуют, что при высокой производительности время развертывания измеряется несколькими минутами или часами, в то время как при низкой производительности — месяцами.

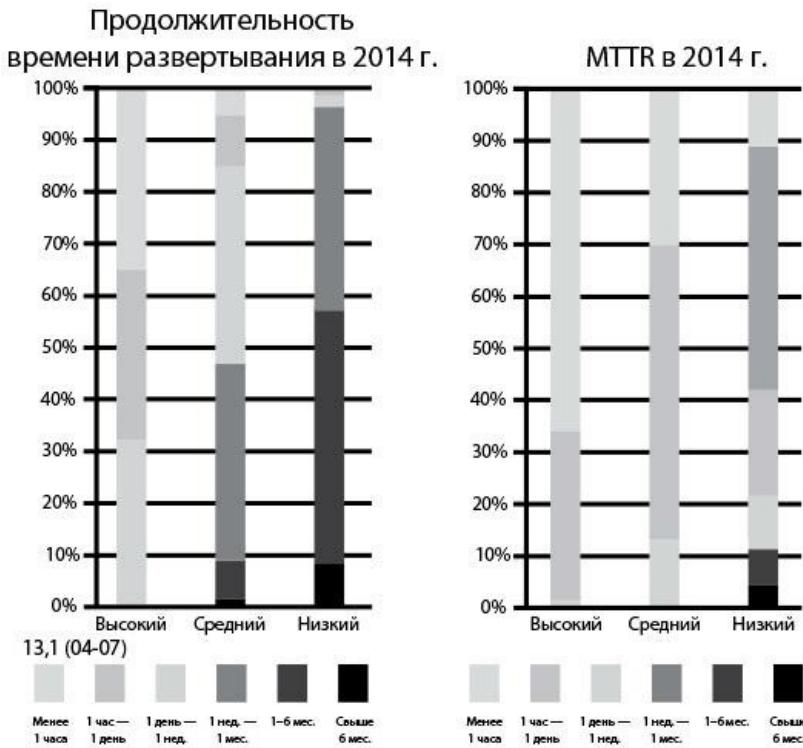


Рис. 18. Компании с высокой производительностью выполняют развертывание гораздо быстрее и также гораздо быстрее восстанавливают работу производственных сервисов после инцидентов (источник: доклад 2014 State of DevOps Report, подготовленный компанией Puppet Labs)

Создав такую возможность, мы используем кнопку «развернуть код», позволяющую быстро и безопасно вносить изменения, сделанные в нашем коде и наших средах, в производство с помощью нашего конвейера развертывания.

В отличие от компании Facebook, где развертывание осуществляется релиз-инженерами, развертывание в компании Etsy может выполнять любой, кто захочет, например разработчик, работник отдела эксплуатации или отдела информационной безопасности. Процесс развертывания стал настолько безопасным и обычным, что и новые специалисты сумеют выполнять его в первый рабочий день, так же как члены правления компании и даже их домашние собачки!

Как писал Ноа Сассман, архитектор тестирования компании Etsy, «в восемь утра, когда начинается обычный рабочий день, примерно 15 человек и их псов организуют очередь: все они ждут, чтобы коллективно развернуть до 25 наборов изменений до конца дня».

Инженеры, желающие развернуть код, первым делом заходят в чат, где добавляют себя в очередь развертывания, смотрят на активность в процессе развертывания — кто еще находится в очереди, рассылают широковещательные сообщения о своих действиях и получают помощь от других инженеров, если требуется. Когда наступает очередь инженера выполнить развертывание, он получает уведомление в чате.

Целью компании Etsy было сделать развертывание в производство легким и безопасным, с наименьшим количеством шагов и минимальным количеством формальностей. Еще до того, как разработчик зафиксирует код, он запустит на своей рабочей станции все 4500 модульных тестов, что займет менее одной минуты. Все вызовы внешних систем, таких как базы данных, закрыты заглушками.

После того как инженеры зафиксировали изменения в основной ветке кода, мгновенно запускаются более семи тысяч автоматизированных тестов основной ветке на серверах непрерывной интеграции (CI — continuous integration). Сассман пишет: «Методом проб и ошибок мы пришли к выводу, что примерно 11 минут — это подходящий срок для максимальной продолжительности работы автоматизированных тестов в течение цикла. Это оставляет запас времени на повторный запуск тестов во время развертывания, если что-то сломается, и нужно будет вносить исправления, не выходя слишком далеко за принятый двадцатиминутный лимит времени».

Если бы все тесты проводились последовательно, то, как отмечает Сассман, «7000 тестов основной ветки потребовали бы около полчаса для своего выполнения. Поэтому мы делим эти тесты на подмножества и распределяем их по 10 серверам в нашем кластере Jenkins [CI]... Разделение тестов

на наборы и параллельный их запуск дают нам нужный срок выполнения — 11 минут».

Следующими надо запустить smoke-тесты — тесты системного уровня, запускающие cURL для выполнения тестовых наборов PHPUnit. После этих тестов запускаются функциональные тесты, выполняющие сквозное тестирование GUI на рабочем сервере, содержащем либо тестовую среду, либо предпроизводственную среду (в компании ее называют «принцесса»), фактически представляющую собой производственный сервер, изъятый из ротации, что обеспечивает его точное соответствие производственной среде.

Эрик Кастрнер пишет, что, после того как наступает очередь инженера выполнять развертывание, «Вы переходите к программе Deployinator (разработанный в компании инструмент, рис. 19) и нажимаете кнопку, чтобы перевести его в режим тестирования. В этом режиме она посещает «принцессу»... затем, когда код готов к работе в реальной производственной среде, вы нажимаете кнопку Prod, и вскоре ваш код работает в производстве, и все в IRC [канале чата] знают, кто выпустил какой код и опубликовал ссылку на список внесенных изменений. Те, кто отсутствует в чате, получают сообщение по электронной почте с той же информацией».

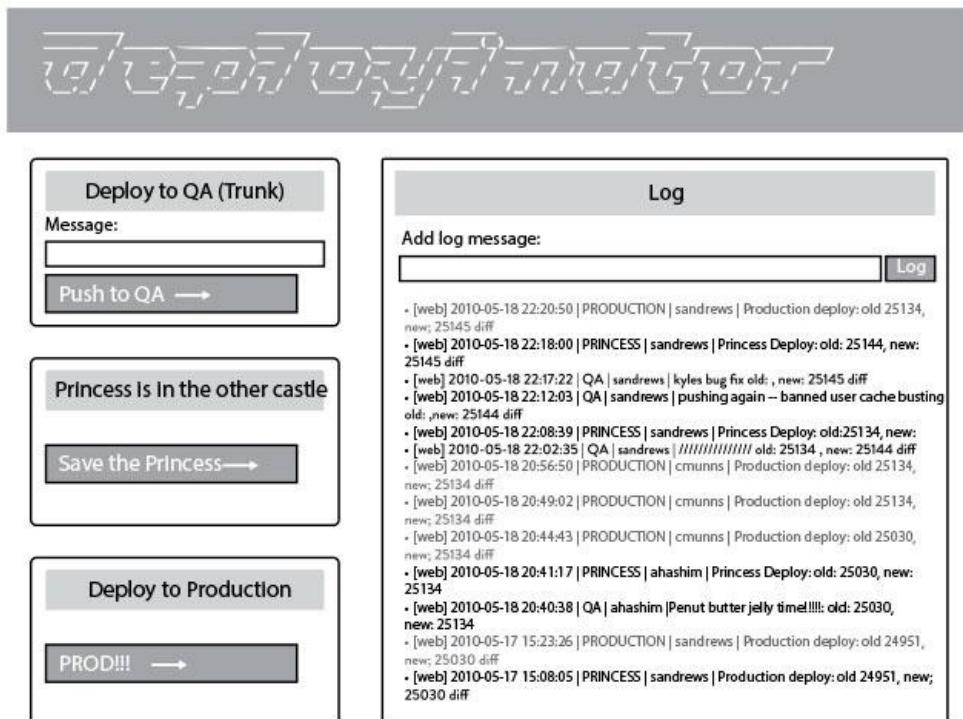


Рис. 19. Консоль программы Deployinator компании Etsy (источник: Erik Kastner, статья Quantum of Deployment на сайте , 20 мая 2010 г., )

В 2009 г. процесс развертывания Etsy вызывал стресс и страх. К 2011 г. он стал обычной операцией, выполняемой от 25 до 50 раз в день, помогая инженерам быстро передать их код в производство, предоставляя ценность клиентам.

При традиционном запуске проекта разработки программного обеспечения даты релизов определяются нашими маркетинговыми сроками запуска в производство. Накануне вечером мы выполняем развертывание готового программного обеспечения (или настолько близкого к полной готовности, насколько мы смогли этого добиться) в производственную среду. На следующее утро мы объявляем о новых возможностях всему миру, начинаем принимать заказы, обеспечиваем клиентам новые возможности и так далее.

Вместе с тем слишком часто бывает, что дела идут не по плану. Мы можем столкнуться с такими производственными нагрузками, которые никогда не тестировали или даже вовсе их не предполагали. Они серьезно затруднят работу сервиса как для клиентов, так и для самой организации. Что еще хуже, для восстановления сервиса может потребоваться непростой процесс отката или в равной мере рискованная операция fix forward, когда мы вносим изменения непосредственно в производство, а это действительно плохой опыт для работников. Когда все наконец заработает, все сотрудники вздыхают с облегчением и радуются, что релизы ПО и их развертывание в производство происходят нечасто.

Конечно, мы знаем, что для достижения желаемого результата — гладкого и быстрого потока —

необходимо выполнять развертывание чаще. Для этого нужно отделить развертывание в производственной среде от релиза версий. На практике слова «развертывание» и «релиз» часто используются как синонимы. Вместе с тем это два различных действия, служащие двум различным целям:

- развертывание — это установка заданной версии программного обеспечения в конкретной среде (например, развертывание кода в среде интеграционного тестирования или развертывание кода в производственной среде). В частности, развертывание может быть, а может и не быть связано с релизом новой функции для клиентов;
- релиз — это когда мы делаем функциональную возможность (или набор возможностей) доступной для всех наших клиентов или клиентам из определенного сегмента (например, мы включаем эту возможность для использования пятью процентами наших клиентов). Наши код и среды должны быть спроектированы таким образом, чтобы при релизе функциональности не требовалось изменение кода нашего приложения.

Другими словами, если мы объединяем понятия развертывания и релиза, это затрудняет оценку успешности результатов, а разделение двух действий усиливает ответственность разработчиков и эксплуатации за успех быстрых и частых развертываний, позволяя владельцам продукта отвечать за успешность бизнес-результатов релизов (стоили ли создание и запуск функциональной возможности затраченного времени).

Методы, описанные выше, обеспечивают нам возможность быстрых и частых развертываний в производство созданных разработчиками функциональных возможностей, имея целью сокращение рисков и последствий ошибок развертывания. Остается риск релиза, то есть вопрос, обеспечивают ли функции, запущенные в производство, достижение желаемых заказчиком и нами результатов в бизнесе.

Если развертывание занимает много времени, это определяет, как часто мы можем выпускать на рынок новые возможности нашего ПО. Однако, когда мы получаем возможность выполнять развертывание по требованию, решение, как часто открывать новые возможности для клиентов, становится маркетинговым и бизнес-решением, а не техническим. Существуют две широкие категории принципов выполнения релизов, и мы можем их использовать:

- **шаблоны релиза на основе среды:** в этом случае мы имеем две или более сред, где выполняется развертывание, но только одна из них получает реальный трафик от клиентов (например, благодаря настройкам средств балансировки нагрузки). Новый код развертывается в среде без реальной нагрузки, и релиз будет произведен перенаправлением трафика к этой среде. Это исключительно эффективные шаблоны, поскольку они обычно не требуют внесения каких-либо изменений в наши приложения. Они включают в себя *сине-зеленое (blue-green) развертывание*, *канареечное развертывание* и *клUSTERНЫЕ иммУнныЕ сИСТЕМЫ (cluster immune systems)*, все они будут обсуждаться ниже;
- **шаблоны релизов на основе приложений:** в этом случае мы изменяем наше приложение, с тем чтобы мы могли выборочно разблокировать и выпускать отдельные функциональности приложения путем небольших изменений конфигурации. Например, мы можем реализовать флаги функций, чтобы постепенно открывать новые возможности в производственной среде для команды разработчиков, или всех наших сотрудников, или 1 % наших клиентов, или, когда мы уверены в том, что релиз будет работать, как запланировано, для всей нашей клиентской базы. Как обсуждалось ранее, это делает возможным использование метода, называемого теневым запуском, когда мы запускаем в производственной среде все функциональные возможности и проверяем их на клиентском трафике до релиза. Например, мы можем скрытно проверить новую функциональность на реальном трафике в течение нескольких недель перед запуском для выявления проблем, чтобы их можно было исправить до официального запуска.

Отделение развертывания от релизов значительно изменяет порядок работы. Нам больше не придется выполнять развертывание в ночное время или в выходные дни, чтобы уменьшить риск негативного влияния на работу клиентов. Вместо этого мы можем выполнить развертывание в обычное рабочее время, давая возможность отделу эксплуатации работать тогда же, когда и все другие сотрудники.

В этом разделе основное внимание уделяется шаблонам релиза на основе среды, не требующим внесения изменений в код приложения. Мы можем сделать это за счет нескольких сред, в которых производится развертывание, но только одна получает реальный трафик клиентов. Тем самым мы

значительно уменьшаем риск, связанный с передачей релизов в производство, и время развертывания.

Наиболее простой шаблон из трех называется сине-зеленым развертыванием. В этом шаблоне есть две производственные среды: синяя и зеленая. В любое время только одна обслуживает трафик клиента, на рис. 20 рабочая среда — зеленая.



Рис. 20. Шаблон сине-зеленого развертывания (источник: Джез Хамбл, Дэвид Фарли «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ»)

Для релиза новой версии сервиса мы развертываем его в неактивную среду, где можно выполнить тестирование, не прерывая работу пользователей. Когда мы уверены в том, что все работает нормально, то выполняем релиз, направляя трафик к голубой среде. Таким образом, эта среда становится рабочей, а зеленая — подготовительной. Откат можно выполнить перенаправлением трафика опять к зеленой среде.

Сине-зеленая схема развертывания — простой шаблон, его также чрезвычайно легко встроить в существующие системы. Он имеет огромные преимущества, например возможность выполнить развертывания в рабочее время и провести простые переналадки (например, изменение настроек маршрутизатора, изменение символьных ссылок) в часы непиковых нагрузок. Одно это может существенно улучшить условия работы команды, выполняющей развертывание.

Проблему создает наличие двух версий приложения в производстве, зависящих от общей базы данных: когда при развертывании требуется изменить схему базы данных или в самой базе добавить, изменить либо удалить таблицу или колонку, она не в состоянии поддерживать обе версии приложения. Есть два общих подхода к решению:

- **создание двух баз данных (например, синей и зеленой):** каждая версия приложения — синяя (старая) и зеленая (новая) — имеет собственную базу данных. В процессе релиза мы ставим синюю базу данных в режим «только для чтения», выполняем ее резервное копирование, восстанавливаем данные в зеленую базу данных и, наконец, переключаем трафик на зеленую среду. Проблема этой модели в том, что, когда нам необходимо вернуться к синей версии, мы потенциально можем потерять транзакции, если только сначала не перенесем их вручную из зеленой версии;
- **отделение изменений базы данных от изменений приложения:** вместо того чтобы поддерживать две базы данных, мы отделяем релиз изменений в базе данных от релиза изменений в приложении, сделав две вещи. Во-первых, мы делаем только дополнения к нашей базе данных и никогда не видоизменяем существующие объекты базы данных, во-вторых, мы не делаем никаких предположений в нашем приложении о том, какая версия базы данных будет работать в производстве. Это сильно отличается от традиционного отношения к базам данных, то есть установки на то, чтобы избегать дупликации данных. Процесс отделения изменений базы данных от изменений приложения был использован компанией IMVU (и другими) приблизительно в 2009 г., что позволило ей делать 50 развертываний в день, причем некоторые из них требовали изменения базы данных.

Джез Хамбл и Дэйв Фарли, соавторы книги «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ», работали над проектом для компании Dixons Retail, большой британской розничной торговой компании, использовавшей тысячи POS-систем, находящихся в сотнях магазинов и работавших под несколькими торговыми марками.

Хотя Blue-Green развертывание в основном ассоциируется с онлайновыми веб-сервисами, Норт и Фарли использовали этот шаблон, чтобы значительно снизить риски и потери времени на переналадку при обновлении ПО POS-терминалов.

Обычно модернизация систем POS производится водопадным методом, как «большой взрыв»: POS-терминалы клиентов и централизованный сервер обновляются одновременно, что требует

продолжительного выключения системы (часто на все выходные дни), а также значительной полосы пропускания сети, чтобы «протолкнуть» новое клиентское программное обеспечение во все розничные магазины. Если не получится провести модернизацию строго по плану, это может очень серьезно нарушить работу магазинов.

Если пропускная способность сети недостаточна для одновременного обновления всех POS-систем, то традиционная стратегия невозможна. Чтобы решить эту проблему, была использована Blue-Green стратегия и созданы две производственные версии централизованного серверного программного обеспечения, чтобы обеспечить возможность одновременной поддержки старой и новой версий POS-терминалов клиентов.

После того как это было сделано, за несколько недель до запланированного обновления POS-терминалов установщики новых версий клиентского программного обеспечения начали отправлять новое ПО в розничные магазины по медленным сетевым каналам и затем развертывать, пока POS-системы находились в неактивном состоянии. В то же время старые версии постоянно работали в нормальном режиме.

Когда все POS-терминалы клиентов были подготовлены к обновлению (обновленный клиент и сервер были успешно протестированы вместе, и новое клиентское программное обеспечение было развернуто у всех клиентов), менеджерам магазинов было предоставлено право решить, когда переходить на новую версию.

В зависимости от потребностей бизнеса некоторые менеджеры хотели бы использовать новые функциональные возможности немедленно, в то время как другие предпочитали подождать. В любом случае, независимо от того, выпускались ли новые функциональные возможности в работу немедленно или после ожидания, менеджерам это было гораздо удобнее по сравнению с централизованным выбором отделом эксплуатации единой даты обновления для всех.

Результатом был значительно более плавный и быстрый релиз ПО, большая удовлетворенность менеджеров магазинов и гораздо меньшие нарушения обычного хода торговых операций. Кроме того, это применение Blue-Green релиза для приложений в виде толстых клиентов ПК демонстрирует, как методы DevOps могут универсально применяться в различных технологиях, зачастую удивительными способами, но с теми же фантастическими результатами.

Шаблон Blue-Green релиза прост в реализации и может значительно повысить безопасность релиза программного обеспечения. Существуют варианты этого шаблона, и они могут способствовать дальнейшему повышению безопасности и сокращению времени развертывания с помощью автоматизации, но за это потенциально придется расплачиваться дополнительным усложнением процесса.

*Шаблон канареичного релиза* автоматизирует процесс релиза, распространяя его последовательно на все более крупные и более критически важные среды, по мере того как мы убеждаемся в том, что код работает, как задумано.

Термин «канареичный релиз» придуман по аналогии с тем, как шахтеры в угольных шахтах брали с собой канареек в клетках, чтобы обнаруживать опасный уровень угарного газа. Если в шахте скапливалось много угарного газа, то он убивал канарейку до того, как становился опасным для шахтеров, и они успевали спастись.

При использовании этого шаблона, когда мы делаем выпуск, то наблюдаем, как программное обеспечение работает в каждой среде. Когда что-то кажется неправильным, мы выполняем откат, в противном случае мы выполняем развертывание в следующей среде.

На рис. 21 показаны группы сред в компании Facebook, созданные для поддержки этого шаблона релизов:

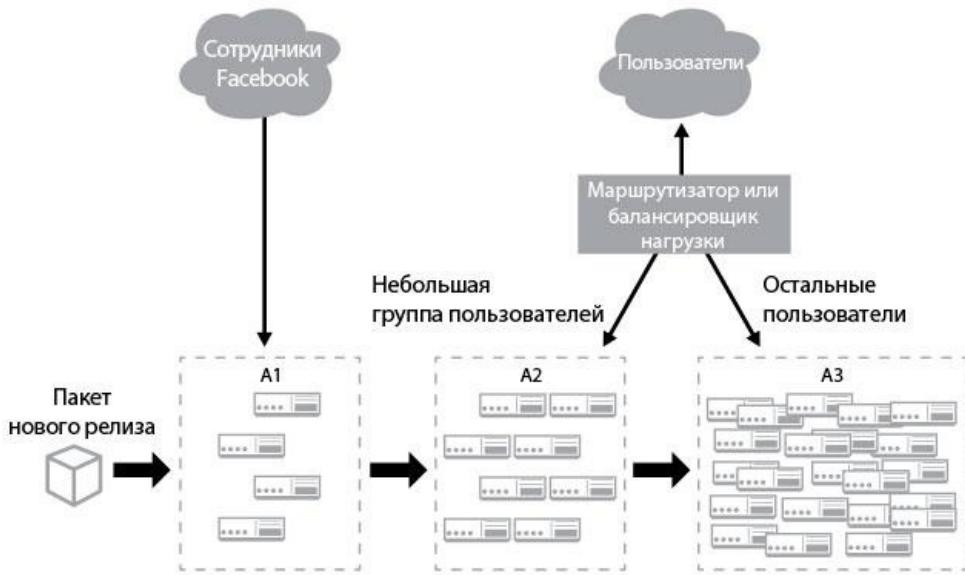


Рис. 21. Шаблон канареичного релиза (источник: Джез Хамбл, Дэвид Фарли «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ»)

- **группа А1:** производственные серверы, обслуживающие лишь внутренних сотрудников;
- **группа А2:** производственные серверы, обслуживающие лишь небольшую долю клиентов. Они развертываются при достижении определенных критериев (автоматически или вручную);
- **группа А3:** остальная часть производственных серверов, развертывающихся после того, как программное обеспечение на серверах в кластере А2 достигнет определенных критериев приемлемости.

Cluster immune system — расширение шаблона канареичного релиза, в ней системы мониторинга производства связываются с процессами релиза и автоматизируется откат на предыдущую версию кода, если производительность системы со стороны пользователя отклоняется от заданных показателей сильнее, чем ожидается. Например, когда коэффициенты пересчета для новых пользователей падают ниже сложившихся показателей на 15–20 %.

Этот тип защиты имеет два существенных преимущества. Во-первых, мы получаем защиту от дефектов. Их трудно найти с помощью автоматизированных тестов, скажем, изменения некоторых важных невидимых элементов веб-страницы (например, изменение CSS). Во-вторых, сокращается время, необходимое для обнаружения того, что в результате изменений снизилась производительность, и реагирования на это.

В предыдущем разделе на основе среды мы создали шаблоны, позволившие отделить развертывание от релиза посредством использования нескольких сред и коммутации потока реальных данных на ту или иную среду, что может быть полностью реализовано на уровне инфраструктуры.

В этом разделе мы опишем шаблоны релиза на основе приложений, которые можем реализовать в нашем коде, обеспечивая еще большую гибкость при безопасном релизе новых возможностей для наших клиентов, зачастую даже выпуская каждую функциональную возможность отдельно. Поскольку шаблоны релиза на основе приложений реализуются в самих приложениях, это требует участия разработчиков в данном процессе.

Основной способ внедрения шаблонов релиза на основе приложений и приложения на основе потребления — реализация переключателей функциональности, предоставляющей механизм для выборочного включения и отключения функции без развертывания кода в производственную среду. С помощью переключателей можно также управлять тем, какие функции видны и доступны конкретным сегментам пользователей (например, внутренним сотрудникам, отдельным сегментам клиентов).

Переключатель функциональности обычно реализуется через управление логикой работы приложений или элементов интерфейса с помощью условного оператора, включающего или отключающего функцию на основе параметра конфигурации, хранящегося где-то в системе. Это может быть просто файл конфигурации приложения (например, файлы конфигурации в форматах

JSON, XML), либо он может быть реализован с помощью службы каталогов или даже веб-службы, специально разработанной для управления функцией переключения.

Переключение функциональных возможностей также дает возможность делать следующее:

- **легко выполнять откат:** функциональные возможности, создающие проблемы или нарушающие работу производственной среды, могут быть быстро и безопасно отключены простым изменением параметра функции переключения. Это особенно ценно, когда развертывания выполняются нечасто — выключение одной конкретной возможности, мешающей какой-то из заинтересованных сторон, обычно намного проще, чем откат всего релиза;
- **ненавязчиво ухудшить производительность:** если наши сервисы испытывают сильную нагрузку, что в обычных условиях требует от нас увеличения мощности оборудования или, что еще хуже, способно привести к сбоям в обслуживании, мы можем использовать переключение функциональных возможностей для снижения качества обслуживания. Другими словами, мы можем увеличить число пользователей, получающих ухудшенную функциональность (например, уменьшаем число клиентов, которые могут получить доступ к определенным возможностям, отключив функциональности, интенсивно использующие процессор, такие как рекомендации, и так далее);
- **увеличить устойчивость нашей службы благодаря использованию сервис-ориентированной архитектуры:** если у нас есть функциональность, опирающаяся на другой сервис, разработка которой еще не завершена, то мы все же можем развернуть нашу новую функциональность в производстве, но скрыть ее с помощью переключателя возможностей. Когда служба станет наконец доступной, мы можем включить эту функциональность. Аналогичным образом, если служба, от которой мы зависим, дает сбой, можно отключить функциональность для предотвращения ее вызовов клиентами, сохраняя в то же время остальную часть приложения работающей.

Чтобы мы могли находить ошибки в функциональных возможностях, использующих переключатели функциональности, автоматизированные приемочные тесты должны работать, когда включена полная функциональность (мы должны также убедиться, что наши переключатели функциональностей работают правильно!).

Переключатели функциональности позволяют выполнять раздельно развертывание кода и релиз новых функций. Далее в книге мы используем его для разработки, управляемой гипотезами, и А/В-тестирования, еще более развивая нашу способность достичь желаемых результатов в бизнесе.

Переключатели функциональности позволяют развертывать новые функции в производственную среду, не делая их доступными для пользователей, то есть применяя метод, известный как *теневой запуск*. В этом случае мы развертываем все функциональные возможности в производственную среду и затем выполняем тестирование их работоспособности, пока они невидимы для клиентов. В случае крупных или рискованных изменений мы часто делаем развертывание за несколько недель до запуска в производственную среду, что дает нам возможность безопасного тестирования с ожидаемыми нагрузками, близкими к производственным.

Предположим, мы выполнили теневой запуск новой функциональной возможности. Она представляет собой значительный риск при релизе — это могут быть новые возможности поиска, процессы создания учетной записи или новые запросы к базе данных. После того как весь код находится в производственной среде, а новая функциональность отключена, мы можем изменить код сессии пользователя, чтобы он выполнял вызовы новых возможностей, однако, вместо того чтобы отображать пользователю результаты этих функций, мы журналируем их или просто отбрасываем.

Например, можно включить одному проценту наших пользователей, работающих в данный момент с сайтом, невидимые вызовы новой функциональности, которую мы собираемся зарелизить, чтобы увидеть, как она работает под нагрузкой. Выявив проблемы и устранив их, постепенно увеличим имитацию нагрузки за счет увеличения частоты вызовов и количества пользователей, проверяющих новую функциональность. Сделав это, можно безопасно имитировать нагрузки, близкие к производственным, и получить уверенность, что наш сервис будет работать так, как и должен.

Кроме того, запуская функциональность, мы можем постепенно выкатывать ее для небольших сегментов клиентов, приостанавливая релиз при обнаружении каких-либо проблем. Так мы сводим к минимуму число клиентов, получающих доступ к функциональности и сразу видящих, что она исчезла, если мы находим дефект или не в состоянии поддерживать требуемый уровень производительности.

В 2009 г., когда Джон Олспоу был вице-президентом отдела эксплуатации компании Flickr, он писал руководству компании Yahoo! о процессе теневого запуска: при таком процессе «увеличивается уверенность каждого работника почти до безразличия, если говорить о страхе перед возникновением проблем, связанных с нагрузкой на сервис. Я не имею представления, сколько развертываний кода в производство было выполнено в любой из дней в течение последних пяти лет, ведь в большинстве случаев я не беспокоился об этом, поскольку эти изменения сопровождались очень низкой вероятностью появления каких-либо нежелательных последствий. Если эти последствия все же проявлялись, любой из работников Flickr мог найти на веб-странице сведения о том, когда были внесены изменения, кто их внес, и точно узнать (строчка за строчкой), что именно было изменено».

Позднее, создав отвечающую нашим требованиям телеметрию в наших приложениях и средах, мы сможем обеспечить более быструю обратную связь для проверки сделанных нами предположений и бизнес-результатов сразу после развертывания функциональности в производство.

При этом не надо ждать, пока произойдет релиз в стиле «большого взрыва», чтобы проверить, действительно ли клиенты хотели бы использовать созданные нами функциональные возможности. Вместо этого к моменту объявления о релизе крупного обновления мы уже проверили бизнес-гипотезы и выполнили бесчисленное множество экспериментов по постоянному совершенствованию продукта с реальными клиентами, что помогло подтвердить: функциональности будут способствовать достижению клиентом желаемых результатов.

На протяжении почти десятилетия Facebook был одним из наиболее широко посещаемых интернет-сайтов по критериям числа просмотренных страниц и уникальных пользователей сайта. В 2008 г. он насчитывал более 70 миллионов активных пользователей, ежедневно посещающих сайт, что создало определенные проблемы для группы, разрабатывающей новую функциональность — чат Facebook.

Евгений Летучий, инженер команды, разрабатывавшей чат, писал о том, как количество одновременных пользователей создало огромную проблему для разработчиков ПО: «Наиболее ресурсоемкой операцией, выполнявшейся в системе чата, была отнюдь не отправка сообщений. Нет, это было отслеживание для каждого пользователя состояния всех его друзей — “в сети”, “нет на месте”, “не в сети”, — чтобы можно было начать разговор».

Реализация этой требующей больших вычислительных мощностей функции было одним из крупнейших технических начинаний за всю историю Facebook, она заняла почти год. Сложность проекта была отчасти обусловлена широким набором технологий, необходимых для достижения требуемой производительности, в том числе C++, JavaScript и PHP, а также тем, что они впервые использовали в серверной инфраструктуре язык Erlang.

После года энергичной работы команда разработки чата зафиксировала свой код в системе контроля версий, после чего он стал развертываться в производство по крайней мере один раз в день. Сначала функциональность чата была видна только команде чата. Позднее она стала видимой для всех внутренних сотрудников компании, но была полностью скрыта от внешних пользователей Facebook с помощью Gatekeeper, службы переключения функций компании Facebook.

В рамках теневого запуска каждый пользовательский сеанс Facebook, запускающий JavaScript в пользовательском браузере, загружал в него тестовую программу: элементы пользовательского интерфейса чата были скрыты, но клиент-браузер мог посыпать невидимые сообщения тестового чата на уже развернутый в производственной среде сервис, позволяя имитировать производственные нагрузки во всем проекте, находить и устранять проблемы с производительностью задолго до выпуска клиентского релиза.

При этом каждый пользователь Facebook — участник программы массового нагружочного тестирования, позволившей команде обрести уверенность, что системы могут обрабатывать реальные производственные нагрузки. Релиз чата и запуск его в производство требовали только двух действий: изменения настроек конфигурации Gatekeeper, чтобы сделать функцию чата видной некоторой части внешних пользователей, и загрузки пользователями Facebook нового кода JavaScript, обрабатывающего UI-чат и отключающего невидимое средство тестирования. Если бы что-то пошло не так, двух шагов было бы достаточно для отката изменений. Когда наступил день запуска чата Facebook, все прошло удивительно успешно и спокойно: без особых усилий чат был масштабирован от нуля до 70 миллионов пользователей за одну ночь. В процессе релиза функциональность чата постепенно включалась для все большего количества пользователей, сначала для внутренних сотрудников Facebook, затем для 1 % клиентов, затем для 5 % и так далее. Как писал Летучий, «секрет перехода от нуля к семидесяти миллионам пользователей за одну ночь — ничего не делать с наскоком».

В уже упоминавшейся книге «Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ» Джез Хамбл и Дэвид Фарли дали определение термина «непрерывная доставка». Термин «непрерывное развертывание» был впервые упомянут

Тимом Фицем в записи блога «Непрерывное развертывание в IMVU: делаем невозможное — 50 развертываний в день». Однако в 2015 г. в ходе работы над этой книгой Джез Хамбл прокомментировал: «В последние пять лет наблюдается путаница вокруг терминов “непрерывная доставка” и “непрерывное развертывание”, и, безусловно, мои собственные идеи и определения изменились со времени написания книги. Каждая организация должна создавать свои варианты методов, исходя из того, в чем она нуждается. Область нашего внимания, ключевые моменты — не форма, а результаты: развертывание должно проходить с низким уровнем риска и запускаться одним нажатием кнопки по требованию».

Его обновленные определения непрерывной доставки и непрерывного развертывания звучат так:

«Когда все разработчики работают с небольшими заданиями в основной ветке кода или все работают вне основной ветки над недолго живущими ветками функций, регулярно объединяемыми с основной, и когда основная ветка постоянно поддерживается в состоянии готовности к развертыванию и мы можем делать релизы кода по требованию одним нажатием кнопки в обычное рабочее время, — мы делаем непрерывную доставку. Если разработчики допускают регрессионные ошибки, включающие дефекты, проблемы производительности, вопросы безопасности, перебои в работе и так далее, то они быстро получают обратную связь. Когда такие проблемы обнаруживаются, они немедленно исправляются, так что основная ветка всегда находится в состоянии готовности к развертыванию.

В дополнение к сказанному выше, когда мы развертываем хорошие сборки в производственную среду на регулярной основе и с помощью самообслуживания (развертывание выполняет либо разработчик, либо отдел эксплуатации), это обычно означает, что мы выполняем развертывание в производство по крайней мере один раз в день на каждого разработчика или, возможно, даже автоматически развертываем каждое изменение кода, зафиксированное разработчиком, — все это означает, что мы вовлечены в режим непрерывного развертывания.

Определенная таким образом непрерывная доставка — необходимое условие для обеспечения непрерывного развертывания, так же как непрерывная интеграция — необходимое условие для непрерывной доставки. Непрерывное развертывание удобно преимущественно в контексте веб-сервисов, предоставляемых через интернет. Однако непрерывная доставка может применяться почти в любом контексте, где мы хотим получить развертывания и релизы высокого качества, быстро выполняемые и обеспечивающие весьма предсказуемые результаты с низким риском, в том числе встраиваемые системы, готовые коммерческие компоненты и мобильные приложения.

В компаниях Amazon и Google большинство команд практикуют непрерывную доставку, хотя некоторые выполняют непрерывное развертывание: имеются существенные различия между командами в том, как часто они развертывают код и каким образом выполняются развертывания. Команды имеют право на выбор способа развертывания исходя из управляемых рисков. Например, команда Google App Engine часто выполняет развертывание один раз в день, в то время как Google Search — несколько раз в неделю.

Аналогично в большинстве примеров, представленных в этой книге, используется непрерывная доставка: это разработка встроенного программного обеспечения принтеров HP LaserJet, система печати счетов компании CSG, работающая на 20 технологических платформах, включая приложения для мейнфреймов, написанные на языке COBOL, разработки компаний Facebook и Etsy. Эти же шаблоны могут использоваться для разработки программного обеспечения, работающего на мобильных телефонах, наземных станциях управления космическими спутниками и так далее.

Как показывают приведенные выше примеры из деятельности компаний Facebook, Etsy и CSG, релизы и развертывания не обязательно должны иметь высокую степень напряженности и требовать драматических усилий десятков или сотен инженеров, работающих в круглосуточном режиме, чтобы завершить эти операции. Вместо этого они могут быть рутинными как часть повседневной работы.

При этом мы можем сократить время развертывания с месяца до нескольких минут, позволяя нашим организациям быстро доставлять ценность клиентам, не вызывая хаоса и сбоев в работе. Кроме того, обеспечив совместную работу разработчиков и отдела эксплуатации, мы можем в конечном счете сделать условия работы этого отдела более гуманными.

## Глава 13. Архитектура низкорисковых релизов

Почти все известные примеры применения DevOps, используемые в качестве образцов, были обречены из-за проблем архитектуры. Об этом уже говорилось в историях LinkedIn, Google, eBay, Amazon и Etsy. В каждом случае компании смогли успешно прийти к архитектурам, более соответствующим текущим организационным потребностям.

Это принцип эволюционной архитектуры: Джез Хамбл отмечает, что архитектура «всех успешных продуктов или организаций обязательно развивается в ходе жизненного цикла». До перехода в Google Рэнди Шуп в 2004–2011 гг. работал главным инженером и ведущим архитектором eBay. Он отмечает, что «и eBay, и Google находятся в процессе пятой полной переработки архитектуры ПО сверху донизу».

Он размышляет: «Оглядываясь назад, я понимаю, что некоторые технологии и выбор архитектуры выглядят замечательными, а другие — неперспективными. Весьма вероятно, что каждое решение в свое время наилучшим образом служило целям организации. Если бы мы в 1995 г. попытались осуществить что-то эквивалентное микросервисам, то мы, скорее всего, потерпели бы неудачу, проект разрушился бы под собственной тяжестью и, возможно, вместе со всей компанией и с нами».

Задача в том, чтобы обеспечивать переход с имеющейся у нас архитектуры на необходимую. В случае с eBay, когда компании надо было переделать архитектуру, они сначала делали небольшой экспериментальный проект, чтобы показать, что они понимают проблемы достаточно хорошо, чтобы предпринять необходимые меры. Например, когда команда Рэнди Шупа в 2006 г. планировала переделать некоторые части сайта на полный набор языка Java, она искала область, дающую наибольший эффект в обмен на вложенные деньги, и с этой целью отсортировала страницы сайта по величине дохода, приносимой каждой из них. Она выбрала области с самыми большими доходами, остановив отбор, когда бизнес-отдача от очередных страниц оказалась недостаточно большой, чтобы оправдать затраченные усилия.

То, что команда Шупа сделала в eBay, хрестоматийный пример эволюционного проектирования с использованием метода, называющегося «удушающее приложение» (strangler application): вместо «вырезания» старого сервиса с архитектурой, больше не помогающей достичь наших организационных целей, и замещения его новым кодом мы скрываем существующую функциональность за API, чтобы избежать дальнейших изменений. Вся новая функциональность затем реализуется в новых сервисах, использующих новую желаемую архитектуру и выполняющих при необходимости вызовы старой системы.

Шаблон удушающего приложения особенно полезен для облегчения миграции отдельных частей монолитного приложения или сильно связанных сервисов в слабо связанные. Слишком часто нам приходится работать в рамках архитектуры с сильно связанными и переплетенными друг с другом частями, нередко созданной несколько лет (или десятилетий) назад.

Последствия слишком сильно связанной архитектуры очень легко заметить: каждый раз, когда мы пытаемся зафиксировать код в основной ветке или зарелизить код в производственную среду, мы рискуем вызвать глобальные сбои (например, мы можем нарушить выполнение всех остальных тестов, функциональность сайта или даже вообще его работу). Каждое небольшое изменение требует огромных объемов работы по согласованию и координации, выполняемой несколько дней или недель, а также одобрения каждой команды, которую потенциально могут затронуть эти изменения. Разворачивание становится проблематичным, количество изменений, объединяемых в один пакет для развертывания, растет, что еще сильнее усложняет интеграцию и увеличивает усилия, необходимые для тестирования, а это повышает и без того уже высокую вероятность, что что-то пойдет не так.

Даже развертывание небольших изменений может потребовать координации с сотнями (или даже тысячами) других разработчиков, и взаимодействие с любым из них может вызвать катастрофический сбой, потенциально требующий нескольких недель на поиск и устранение проблемы (это приводит к появлению другого симптома: «Разработчики используют только 15 % своего времени на создание кода, а остальная часть тратится на совещания»).

Все эти факторы способствуют использованию крайне небезопасных систем работы, когда небольшие изменения могут вызвать неожиданные и катастрофические последствия. Они также часто способствуют развитию страха перед процессами интеграции и развертывания нашего кода и укрепляют нисходящую спираль, вызывающую стремление выполнять развертывание менее часто.

Исходя из перспективы развития корпоративной архитектуры, нисходящая спираль — следствие второго закона архитектурной термодинамики, особенно в больших и сложных организациях. Чарльз Бец, автор книги *Architecture and Patterns for IT Service Management, Resource Planning, and Governance: Making Shoes for the Cobbler's Children*, отмечает: «Владельцы IT-проектов не несут

ответственности за свой вклад в общую систему энтропии». Другими словами, сокращение общей сложности и увеличение продуктивности всех наших команд редко становится целью отдельного проекта.

В этой главе мы опишем шаги, которые можно предпринять, чтобы повернуть нисходящую спираль вспять, рассмотрим основные архитектурные архетипы, изучим характерные черты архитектуры, позволяющие повысить продуктивность работы разработчиков, тестируемость, развертываемость и безопасность, а также оценим стратегии, дающие возможность осуществить безопасную миграцию с текущей архитектуры, какой бы она ни была, на ту, что лучше обеспечивает достижение наших организационных целей.

В отличие от сильно связанной архитектуры, препятствующей производительности сотрудников и их способности безопасно вносить изменения, слабо связанная архитектура с хорошо определенными интерфейсами, повышающими возможность модулей взаимодействовать друг с другом, способствует повышению производительности и безопасности. Она позволяет небольшим, продуктивным, «двутищевым» командам создавать небольшие изменения, разворачиваемые безопасно и независимо от других команд. И поскольку каждый такой сервис имеет также хорошо определенный API, он позволяет легче тестировать сервисы и создавать контракты и соглашения об уровне обслуживания между командами.

- Облачное хранилище данных компании Google
  - Облачное хранилище данных: NoSQL-сервис
  - Хорошо масштабируемый и устойчивый
  - Высокая согласованность транзакций
  - Богатые возможности запросов, сходные с SQL
- Мегахранилище: структурированная база данных с геометками
  - Многострочные транзакции
  - Синхронная репликация между data-центрами
- Bigtable: структурированное хранилище на уровне кластеров
  - (строка, столбец, метка времени) -> содержимое ячейки
- Colossus: новое поколение кластеризованной файловой системы
  - Блочные распределение и репликация
- Инфраструктура управления кластером
  - Планировщик задач, назначение машин

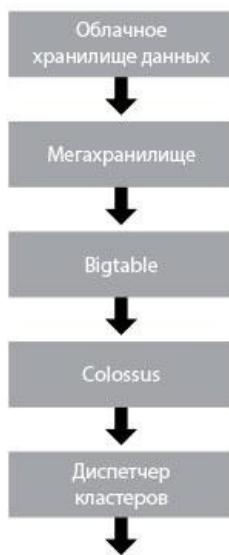


Рис. 22. Облачное хранилище данных компании Google (источник: Shoup, From the Monolith to Micro-services)

Согласно описанию Рэнди Шупа, «этот тип архитектуры отлично служил Google, в частности для такой службы, как Gmail и еще пяти или шести других сервисов более низкого уровня. Их она использовала, и каждый базировался на весьма специфических функциях. Любая служба поддерживается небольшой командой, создающей и запускающей функциональность этой службы, и каждая команда потенциально может использовать различные технологии. Другой пример — служба облачного хранилища данных Google, один из крупнейших NoSQL-сервисов в мире. Несмотря на это, она поддерживается командой из восьми человек. Это возможно в значительной степени из-за того, что сервис основан на нескольких слоях зависимых служб, построенных друг на друге».

Ориентированная на сервисы архитектура позволяет небольшим командам работать над более мелкими и простыми единицами развертывания. Каждая команда может развертывать их независимо, быстро и безопасно. Шуп отмечает, что «организации с этими типами архитектуры, такие как Google и Amazon, показывают, как это может повлиять на организационные структуры, создавая гибкость и масштабируемость. Обе организации имеют десятки тысяч разработчиков, небольшие команды невероятно продуктивны».

В определенный момент своей истории почти все организации, использующие сейчас DevOps, были отягощены плотно связанной, монолитной архитектурой. Успешно помогая им соответствовать требованиям рынка или требованиям к продуктам, она создавала риск сбоев организационной структуры, если пользователям надо было значительно расширять деятельность (например, монолитное приложение на C++ компании eBay в 2001 г., монолитное приложение OBIDOS компании Amazon в том же году, монолитное клиентское Rails-приложение Twitter в 2009 г. и монолитное приложение Leo компании LinkedIn в 2011 г.). В каждом из этих случаев компании смогли перепроектировать свои системы и заложить основы не только выживания, но и

процветания и победы на рынке.

Монолитные архитектуры неплохи по своей сути, в реальности они часто становятся лучшим выбором для организации на раннем этапе жизненного цикла продукта. Как отмечает Рэнди Шуп, «нет идеальной архитектуры для всех продуктов и всех масштабов. Любая архитектура соответствует некоторому набору целей или диапазонам требований и ограничений, таких как время вывода на рынок, простота развития функциональности, масштабирование и так далее. Функциональность любого продукта или услуги почти наверняка будет изменяться с течением времени, так что нет ничего удивительного, что наши архитектурные потребности будут меняться. Что работает в масштабе 1x, редко работает в масштабе 10x или 100x».

Основные архитектурные архетипы показаны в табл. 3, каждая строка означает различные эволюционные потребности организации, каждый столбец показывает достоинства и недостатки разных архетипов. Как видно из таблицы, монолитная архитектура, поддерживающая стартапы (например, быстрое прототипирование новых функций и возможность отклонений или больших изменений в первоначальных стратегиях), очень отличается от архитектуры, требующей наличия сотен команд разработчиков, когда каждая должна быть в состоянии самостоятельно предоставлять продукт клиентам. Поддерживая эволюционность архитектур, мы можем добиться того, что наша архитектура всегда обслуживает текущие потребности организации.

Таблица 3. Архитектурные архетипы

	<b>Достоинства</b>	<b>Недостатки</b>
Монолитная версии 1 (вся функциональность в одном приложении)	Простая вначале Низкие задержки между процессами Одна база кодов, один блок развертывания Эффективное использование ресурсов при небольших масштабах	Накладные расходы на координацию увеличиваются по мере увеличения размера команды Плохая поддержка модульности Плохое масштабирование Развертывание по принципу «все или ничего» (просто, сбои) Долгое время создания
Монолитная версии 2 (набор монолитных слоев: «представление клиентского уровня», «сервер приложений», слой базы данных)	Простая вначале Легкое присоединение к запросам Единая схема развертывания Эффективное использование ресурсов при небольших масштабах	Тенденция к увеличению внутренних связей с течением времени Плохое масштабирование и избыточность («все или ничего», только вертикальное) Трудно настроить должным образом Схема управления «все или ничего»
Микросервисная (модульная, независимая, связь в виде графов, а не слоев, изолированная живучесть)	Каждый блок по отдельности прост Независимые масштабирование и производительность; Независимые тестирование и развертывание Можно оптимально настраивать производительность (кэширование, репликации и так далее)	Много сотрудничающих подразделений Много небольших репозиториев Требует более сложных инструментальной оснастки и управления зависимостями Сетевые задержки

Источник: Shoup, From the Monolith to Micro-services.

Одно из наиболее изученных преобразований архитектуры произошло в компании Amazon. В интервью с обладателем награды ACM Turing и техническим специалистом компании Microsoft Джимом Греем Вернер Фогельс, технический директор компании Amazon, объяснил, что Amazon.com начал работу в 1996 г. как «монолитное приложение на веб-сервере, обращаясь к базе данных на серверной части. Это приложение, получившее название Obidos, эволюционировало, чтобы сохранить всю бизнес-логику, всю логику отображения и все функциональные возможности, которые в конце концов прославили Amazon: аналоги, рекомендации, Listmania, обзоры и так далее».

С течением времени Obidos разросся и стал слишком запутанным, со сложными взаимосвязями отдельных частей. Его невозможно было масштабировать по мере необходимости. Фогельс рассказал Грею, что это означает: «Многие вещи, которые вы хотели бы видеть работающими в хорошей программной среде, больше не могут быть сделаны; множество сложных единиц программного обеспечения объединены в единую систему, но она больше не могла развиваться».

Описывая процесс обдумывания новой желаемой архитектуры, он рассказывал Грею: «Мы прошли через период серьезного самоанализа и пришли к выводу о том, что сервис-ориентированная архитектура могла бы обеспечить уровень изоляции, достаточный для того, чтобы позволить нам

создавать множество компонентов программного обеспечения быстро и независимо друг от друга».

Фогельс отмечает: «Компания Amazon в течение последних пяти лет (2001–2005) прошла через большие изменения архитектуры, чтобы сменить двухуровневую монолитную архитектуру на полностью распределенную децентрализованную платформу сервисов, обслуживающую множество различных приложений. Потребовалось множество инноваций, чтобы такие изменения стали возможными, и мы были одними из первых, использовавших этот подход». Из опыта работы Фогельса в компании Amazon можно извлечь следующие уроки, имеющие большое значение для нашего понимания смен архитектуры:

- урок 1: при неукоснительном применении строгая ориентация на сервисы — отличный метод для достижения изоляции, вы можете добиться невиданного прежде уровня владения и управления;
- урок 2: запрет прямого доступа клиентов к базе данных делает возможным выполнение масштабирования и повышение надежности вашего сервиса, не затрагивая клиентов;
- урок 3: процессы разработки и эксплуатации получают значительную выгоду от перехода на сервис-ориентированную модель. Она стала ключевым фактором успеха в создании команд, быстро внедряющих инновации в интересах клиентов. Каждая служба имеет команду, несущую полную ответственность — от оценки функциональности до создания архитектуры, разработки и управления ее работой.

Применение этих результатов повышает продуктивность работы и надежность до показателей, захватывающих дух. В 2011 г. компания Amazon выполняла примерно 15 тысяч развертываний в день. К 2015 г. она выполняла почти 136 тысяч развертываний в день.

Термин «удушающее приложение» был введен Мартином Фаулером в 2004 г., после того как он увидел огромные удушающие лозы во время путешествия в Австралию и описал их так: «Они выбирают верхние ветви смоковницы и постепенно растут вниз по дереву, пока не укоренятся в почве. Долгие годы они разрастаются, приобретая фантастически красивые формы и постепенно удушая и убивая дерево, на котором паразитируют».

Определив, что нынешняя архитектура слишком сильно связана, мы можем безопасно запустить отвязывание части функциональности от нашей существующей архитектуры. Это позволит командам, поддерживающим отвязываемые функциональные возможности, самостоятельно разрабатывать, тестировать и развертывать их код в производство, делая это автономно и безопасно и снижая архитектурную энтропию.

Как говорилось ранее, шаблон удушающего приложения включает размещение существующей функциональности за API, где она остается неизменной, новые функции внедряются с помощью нашей желаемой архитектуры, при необходимости выполняя вызовы старой системы. Реализуя удушающее приложение, мы стремимся обеспечить доступ ко всем сервисам через версионированные API, также называемые *версионированными или неизменяемыми сервисами*.

Версионированные API позволяют нам изменить сервис без влияния на вызывающих его клиентов, что дает возможность системе быть более слабо связанной — если нам нужно изменить аргументы, мы создаем новую версию API и переводим команды, зависящие от нашего сервиса, на новую версию. В конце концов мы не сможем достичь цели изменения архитектуры, если позволим новому удушающему приложению стать тесно связанным с другими службами (например, подключаясь непосредственно к базе данных другой службы).

Если вызываемые службы не имеют четко определенных API, то мы должны создавать такие интерфейсы или по крайней мере скрыть сложность взаимодействия с такими системами использованием библиотеки клиента, имеющей четко определенный API.

Неоднократно отвязывая функциональность от существующих тесно связанных систем, мы перемещаем работу в безопасные и активные экосистемы, после чего разработчики могут стать гораздо более продуктивными, а устаревшие приложения уменьшают функциональность. Они даже могут исчезнуть полностью, когда все необходимые функции перейдут к новой архитектуре.

Создав удушающее приложение, мы сможем избежать точного воспроизведения существующей функциональности в некоторых новых архитектурах или технологиях — часто наши бизнес-процессы более сложны, чем необходимо в связи с особенностями имеющихся систем, которые мы реплицируем (изучая пользователя, мы часто можем перепроектировать этот процесс, чтобы создать намного более простые и более рациональные средства для достижения наших бизнес-целей).

Наблюдение Мартина Фаулера подчеркивает этот риск: «Значительную часть моей трудовой деятельности составляло участие в переписывании критически важных систем. Вам может показаться, что это очень просто — написать новую программу, делающую то же самое, что и старая. Но такая работа всегда более сложна, чем кажется, и изобилует всевозможными рисками. Впереди маячит важная дата завершения проекта, и давление нарастает. В то время как новые функциональные возможности (всегда существуют новые функциональности) нравятся всем, старые никуда не деваются. Нередко в переписанную систему приходится добавлять даже старые ошибки».

Как и в случае с любым преобразованием, мы стремимся создать быстрый победный результат и рано доставить дополнительную ценность, еще до того как продолжим итерации. Предварительный анализ помогает нам определить наименьший возможный кусок работы, полезный для получения бизнес-результатов с использованием новой архитектуры.

Компания Blackboard — один из пионеров предоставления технологии для учебных заведений с годовым доходом в размере примерно 650 миллионов долларов в 2011 г. В то время команда разработчиков флагманской программы Learn, пакетного программного обеспечения, установленного и запущенного локально на сайтах клиентов, ежедневно сталкивалась с последствиями использования унаследованной базы кода на J2EE, писавшейся еще в 1997 г. Как отмечал Дэвид Эшман, главный архитектор компании, «мы до сих пор имеем дело с фрагментами кода на языке Perl, встречающимися на многих участках нашего кода».

В 2010 г. Эшман сосредоточился на сложности и растущем времени разработки, связанных со старой системой, отмечая, что «сборка, интеграция и тестирование процессов становились все более сложными и более склонными к ошибкам. И чем больше становился продукт, тем дольше длилась разработка новых функциональных возможностей и тем хуже были результаты, получаемые нашими клиентами. Даже получение обратной связи от процесса интеграции требовало от 24 до 36 часов».

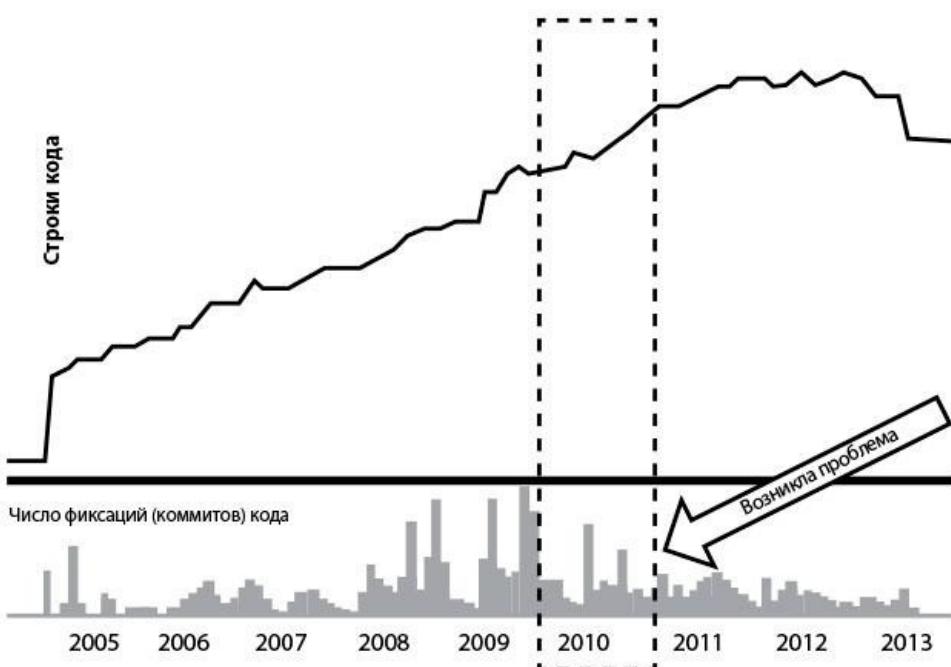


Рис. 23. Репозиторий кода Blackboard Learn до создания Building Blocks (источник: DOES14 — David Ashman — Blackboard Learn — Keep Your Head in the Clouds, видео на YouTube, 30:43, размещено 28 октября 2014 г. оргкомитетом конференции DevOps Enterprise Summit 2014, )

Как это начало влиять на производительность труда разработчиков, Эшман увидел на графиках, созданных на основе данных из репозитория исходного кода начиная с 2005 г.

На рис. 24 верхний график показывает число строк кода в репозитории монолитной программы Blackboard Learn, нижний график показывает число фиксаций кода. Проблема, ставшая для Эшмана очевидной, заключалась в том, что количество фиксаций кода стало уменьшаться, объективно показывая нарастающую трудность внесения изменений в код, в то время как число строк кода продолжало увеличиваться. Эшман отмечал: «Эти графики свидетельствовали, что нам нужно что-то делать, в противном случае проблемы будут усугубляться, и этому не видно конца».

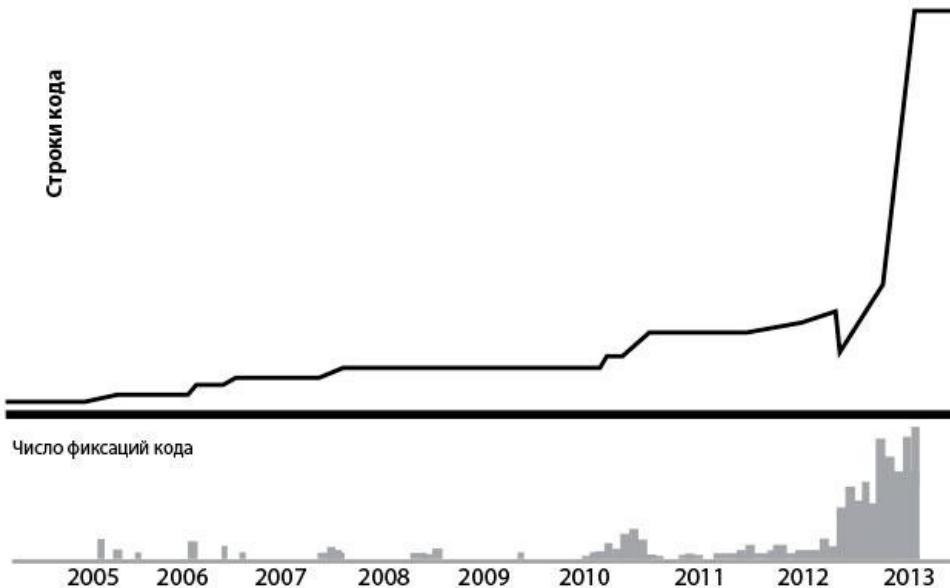


Рис. 24. Репозиторий кода Blackboard Learn с использованием Building Blocks (источник: DOES14 — David Ashman — Blackboard Learn — Keep Your Head in the Clouds, видео на YouTube, 30:43, размещено 28 октября 2014 г. оргкомитетом конференции DevOps Enterprise Summit 2014, )

В результате в 2012 г. Эшман сосредоточился на реализации проекта переработки архитектуры кода, использовавшего шаблон удушающего приложения. Команда достигла этого путем создания инструмента, названного Building Blocks. Он позволил разработчикам работать над отдельными модулями, отделенными от монолитной базы кода. Доступ к ним осуществлялся через фиксированные API. Это позволило командам работать автономнее, без необходимости постоянно общаться друг с другом и координировать свою деятельность с другими командами.

Когда Building Blocks стал доступен разработчикам, размер репозитория монолитного исходного кода начал уменьшаться (измеренный в количестве строк кода). Эшман пояснил: это происходило потому, что разработчики перемещали код в репозиторий исходного кода модулей Building Blocks. «Фактически, — сообщил Эшман, — каждому разработчику, которому предоставлена возможность выбора, хотелось бы работать в базе кода Building Blocks, где они могли бы работать с большей автономностью, свободой и безопасностью».

На графике выше показана связь между экспоненциальным ростом числа строк кода и экспоненциальным ростом числа фиксаций кода в репозитории Building Blocks. Новая база кода Building Blocks дала разработчикам возможность быть более продуктивными, и они сделали работу безопаснее, поскольку ошибки стали приводить к небольшим местным сбоям вместо крупных катастроф, влияющих на глобальную систему.

Эшман заключает: «Когда разработчики начали работать с архитектурой Building Blocks, созданной с целью реализации впечатляющих улучшений в модульности кода, это позволило им работать более независимо и свободно. В сочетании с обновлениями в нашем процессе сборки это обеспечило им возможность быстрее и лучше получать обратную связь, что означает лучшее качество».

Архитектура, в рамках которой работают наши сервисы, в значительной степени диктует то, как мы будем тестировать и развертывать наш код. Это было подтверждено в докладе 2015 State of DevOps Report компании Puppet Labs, показавшем, что архитектура — один из лучших инструментов предсказания производительности инженеров, работающих в ее рамках и того, насколько быстро и безопасно можно производить изменения.

Поскольку мы часто «застреваем» в архитектуре, оптимизированной для другого набора организационных целей или в далеком прошлом, постольку должны иметь возможность безопасно переходить с одной архитектуры на другую. В практических примерах из этой главы, а также в ранее приведенном примере компании Amazon показывались методики (например, шаблон удушающего приложения), способные помочь постепенно мигрировать с одной архитектуры на другую, давая возможность адаптироваться к потребностям организации.

В предыдущих главах третьей части мы рассмотрели архитектуру и технические методы, обеспечивающие быстрый поток работы от разработчиков к отделу эксплуатации, так что продукт может быть доставлен заказчикам быстро и безопасно.

В четвертой части мы будем создавать архитектуру и механизмы, обеспечивающие быстрое протекание процесса обратной связи в противоположном направлении, справа налево, чтобы

быстрее найти и устранить проблемы, распространить обратную связь на всю организацию и обеспечить лучшие результаты нашей работы. Это позволит нашей организации еще больше увеличить скорость адаптации.

## **Введение**

В части III мы описали архитектуру и технические практики, необходимые, чтобы создать быстрый поток из разработки в ИТ-эксплуатации. В части IV мы расскажем о том, как применять технические практики второго пути, чтобы создать быстрый и непрерывный поток обратной связи из ИТ-эксплуатации в разработку.

Применяя эти методики, мы сокращаем и усиливаем петли обратной связи, чтобы фиксировать проблемы в зародыше и распространять информацию о них по всему циклу создания продукта. Это позволяет быстро найти и исправить проблемы в начале цикла разработки программного обеспечения, в идеале — задолго до того, как они приведут к катастрофическим последствиям.

Кроме того, мы создадим такую систему работы, где знания, полученные в ИТ-эксплуатации, встраиваются в обратный поток, ведущий в разработку и управление продукцией. Это позволяет нам быстро внедрять улучшения и новый опыт, полученные из ранних индикаторов проблем или из привычных способов использования продукта нашими клиентами, на стадии разработки, внедрения.

Помимо всего прочего, мы создадим процесс, позволяющий участникам получать обратную связь о своей работе, визуализирующий данные для улучшения выводов и позволяющий нам быстро тестировать связанные с продуктом гипотезы, помогая определить, способствуют ли создаваемые нами средства достижению целей нашей организации.

Мы также покажем, как создать систему измерения показателей, отражающую процессы разработки, тестирования и внедрения, а также пользовательское поведение, проблемы на стадии эксплуатации и аварийные падения программ, проблемы аудита и нарушения требований защищенности. Усиливая сигналы в ходе ежедневной работы, мы можем увидеть и решить проблему в зародыше. Мы создаем системы работы, дающие возможность уверенно вносить изменения и устраивать эксперименты с разрабатываемым продуктом, зная, что можно быстро обнаружить и устранить возникшую проблему. Для реализации этих целей мы рассмотрим следующие пункты:

- создание системы измерения показателей, позволяющей фиксировать и решать проблемы;
- использование этой системы, чтобы лучше предугадывать вероятные проблемы и добиваться поставленных целей;
- интегрирование опыта пользователей и их обратной связи в работу команд разработчиков;
- создание системы обратной связи, чтобы разработка и ИТ-эксплуатация могли безопасно осуществлять внедрение продукта;
- получение обратной связи для улучшения качества работы с помощью оценок коллег и парного программирования.

Шаблоны в этой главе помогают укреплять общие цели подразделений управления продукцией, разработки, контроля качества, ИТ-эксплуатация и информационной безопасности и побуждать их делить ответственность за то, чтобы процесс создания продукта протекал гладко, а также вместе работать над тем, чтобы вся система в целом становилась лучше. Где возможно, мы покажем, к каким следствиям приводят те или иные действия. Чем больше беспочвенных предположений мы сможем развеять, тем быстрее сможем обнаружить и исправить недостатки и тем сильнее будет наша способность обучаться и вносить позитивные изменения.

В следующих главах мы покажем, как встроить в работу петли обратной связи: они позволят всем сотрудникам работать над достижением общих целей, замечать проблемы сразу при их появлении, настраивать быстрое обнаружение сбоев и восстановление, а также проверять, что все элементы функциональности не только работают так, как запланировано, но также соответствуют целям организации и поддерживают получение новых знаний и обмен опытом.

## Глава 14. Создайте телеметрию, позволяющую замечать проблемы и решать их

Суровая реальность жизни отдела ИТ-эксплуатации — не все идет так гладко, как хотелось бы. Небольшие изменения приводят к многообразным и неожиданным последствиям, включая задержки, простой в работе и масштабные сбои, влияющие на всех наших клиентов. Это реальность управления сложными системами; ни один человек не в состоянии охватить взглядом всю систему и понять, как ее части взаимодействуют друг с другом.

Когда в нашей повседневной работе происходит какой-нибудь неожиданный сбой, у нас не всегда есть информация, необходимая для решения проблемы. Например, не всегда можно определить, произошел ли сбой из-за проблемы в нашем приложении (например, из-за дефекта кода), в окружении (например, вследствие проблемы с передачей данных по сетям или с конфигурацией сервера) или в чем-то полностью внешнем (к примеру, DoS-атака).

В подразделении ИТ-эксплуатации с этими проблемами можно бороться с помощью золотого правила: что-то пошло не так — перезагрузи сервер. Если это не сработало, перезагрузи сервер рядом с предыдущим. Если и это не сработало, перезагрузи все серверы. Если же и это не исправило ситуацию, свали все на разработчиков, все сбои ведь происходят из-за них.

В то же время исследование, проведенное Microsoft Operations Framework (MOF) в 2001 г., показало, что организации с самым высоким уровнем обслуживания перезагружали свои серверы в 20 раз реже, чем условная средняя компания, и они в пять раз реже сталкивались с «синим экраном смерти». Другими словами, они обнаружили, что самые успешные компании гораздо лучше диагностировали и исправляли проблемы с серверами, применяя на практике то, что Кевин Бер, Жене Ким и Джордж Спаффорд назвали в своей книге *The Visible Ops Handbook* «культурой причинно-следственных связей». Успешные фирмы последовательно решали проблемы, используя производственную телеметрию, чтобы понять возможные причины неполадок и найти рабочие методы их устранения, в отличие от менее успешных фирм, слепо перезагружавших серверы.

Чтобы образ действий был направлен на решение проблем, нужно выстроить системы на основе непрерывно поступающих телеметрических данных. Под телеметрией обычно понимают «процесс автоматической коммуникации, с помощью которого измерения и другие данные собираются в удаленных точках и затем передаются на приемные устройства для последующего контроля». Наша цель в том, чтобы создать систему телеметрии в наших приложениях и окружении как внутри процесса разработки, так и в процессе развертывания и внедрения.

Майкл Рембетси и Патрик Макдоннелл описали, как наблюдение и контроль над созданием продукции были важнейшей частью начавшегося в 2009 г. перехода к системе DevOps в компании Etsy. Это было связано с тем, что они стандартизовали и перевели весь свой стек технологий в LAMP (Linux, Apache, MySQL и PHP), избавляясь от множества других технологий, поддерживать которые становилось все труднее.

В 2012 г. на конференции Velocity Макдоннелл рассказал, насколько рискованно это было: «Мы меняли нашу важнейшую инфраструктуру, чего клиенты никогда не заметили бы. Однако они точно заметили бы, если бы мы где-то напортачили. Нам нужно было больше показателей, чтобы можно было быть уверенными в том, что мы ничего не ломаем, проводя эти большие перемены. Это было нужно как для наших технических служб, так и для нетехнических, например для отдела маркетинга».

Далее Макдоннелл объясняет: «Мы начали аккумулировать все данные по серверам в систему мониторинга Ganglia, визуализировать ее в Graphite, программе с открытым исходным кодом, в которую мы много инвестировали. Мы собирали все данные в единое целое, начиная с бизнес-показателей и заканчивая количеством развертываний. Тогда мы модифицировали Graphite с помощью того, что называем “нашей уникальной и неповторимой методикой вертикальных линий”, отображающей на все графики показателей моменты, когда происходило развертывание. Благодаря такому подходу мы могли быстро замечать любые непредусмотренные побочные эффекты развертывания и внедрения. Мы даже начали ставить по всему офису мониторы, чтобы все могли видеть, как работают наши системы».

Обеспечивая возможность разработчикам добавлять телеметрию к их повседневной работе, они создали достаточно телеметрических систем, чтобы развертывания стали безопасными. К 2011 г. Etsy отслеживала более 200 тысяч производственных показателей на каждом уровне стека приложений (например, функциональные возможности приложения, работоспособность приложения, базы данных, операционная система, память, сетевые соединения, безопасность и так далее), а 30 наиболее важных показателей постоянно отображались на «информационной панели развертывания». К 2014 г. они отслеживали более 800 тысяч индикаторов, тем самым демонстрируя непреклонное стремление измерить все, что можно, и максимально упростить сам процесс измерения.

Как пошутил один из инженеров Etsy Йен Малпасс, «если в Etsy и есть религия, то это церковь графиков. Если что-то меняется, мы это отслеживаем. Иногда мы рисуем график того, что еще не меняется, на тот случай, если оно вдруг решит пуститься во все тяжкие... Отслеживание всего, что только можно, — ключ к быстрому продвижению вперед, но единственный способ воплотить этот принцип в жизнь — сделать так, чтобы отслеживание показателей было простым... Мы облегчаем сотрудникам задачу наблюдения за нужными им показателями сразу же, без поглощающих время изменений конфигурации или других сложных процессов».

Одним из открытий доклада 2015 State of DevOps Report было то, что в успешных организациях проблемы в процессе создания программного продукта решались в 168 раз быстрее, чем в других фирмах, причем у ведущих организаций медианное значение MTTR измерялось в минутах, тогда как медианный показатель MTTR фирм с низкими показателями измерялся в днях. Две самые важные методики, позволившие минимизировать MTTR, заключались в использовании контроля версий IT-эксплуатацией и применении телеметрии и проактивного мониторинга рабочей среды.

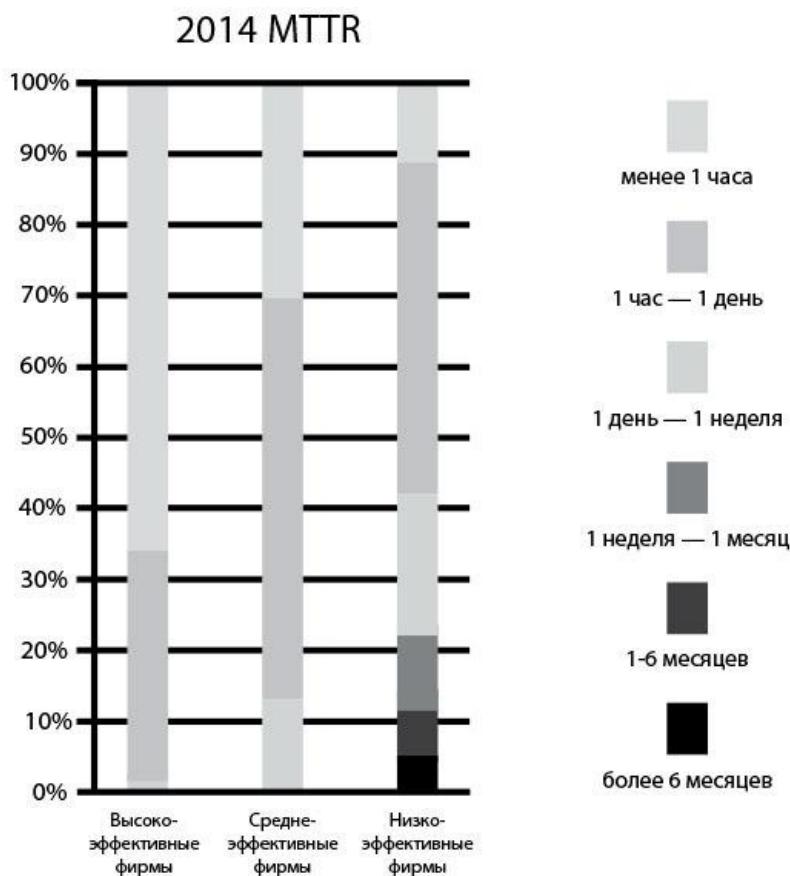


Рис. 25. Время устранения проблемы для высокоеффективных, среднеэффективных и низкоэффективных организаций (источник: отчет 2014 State of DevOps Report компании Puppet Labs)

Наша цель в этой главе — вслед за Etsy построить такую систему телеметрии, чтобы мы всегда были уверены, что наши процессы в производстве протекают так, как нужно. И когда проблемы все-таки случаются, мы можем быстро определить, что пошло не так, и принять обоснованные решения, как лучше всего их разрешить, в идеале — задолго до того, как они повлияют на наших клиентов. Более того, телеметрия как раз и становится средством, позволяющим составить правильное представление о реальности и обнаружить места, где наше понимание расходится с истинным положением дел.

Эксплуатационный мониторинг и логирование — процедуры далеко не новые; многие поколения инженеров эксплуатации использовали и настраивали под себя разные платформы мониторинга (например, HP OpenView, IBM Tivoli, BMC Patrol/BladeLogic), чтобы обеспечить работоспособность производственных систем. Данные обычно собирались с помощью работающих на серверах агентов или с помощью мониторинга без агентов (например, SNMP-ловушки или наблюдение с помощью опросов). Часто для front-end был специальный графический интерфейс (GUI), а отчеты для back-end делались подробнее с помощью таких инструментов, как Crystal Reports.

Точно так же методики разработки приложений с помощью эффективного логирования и

использования результатов телеметрии не есть нечто новое: практически для любого языка программирования существует множество разнообразных проработанных библиотек для логирования.

Однако в течение долгих десятилетий все, что у нас было — лишь разрозненные данные, логи созданные разработчиками, интересные только разработчикам, в эксплуатации же следят только за тем, работает ли среда или нет. В результате, когда происходит сбой, никто не может определить, почему вся система работает не так, как должна, или почему конкретный компонент выходит из строя. В результате, когда происходит сбой, никто не может определить, почему вся система работает не так, как должна, или почему конкретный компонент выходит из строя. Все это сильно осложняет попытки вернуть систему в рабочее состояние.

Чтобы отслеживать проблемы в момент возникновения, необходимо так спроектировать и организовать приложения и среду, чтобы они предоставляли телеметрию, позволяющую понять, как работает система в целом. Когда на всех уровнях стека приложения есть мониторинг и логирование, можно применять другие важные инструменты, например создание графиков и визуализацию показателей, обнаружение аномалий, проактивное оповещение и распространение информации и так далее.

В книге *The Art of Monitoring* Джеймс Торнбулл описал современную архитектуру мониторинга, разработанную и используемую инженерами эксплуатации в масштабных интернет-компаниях (например, Google, Amazon, Facebook). Эта архитектура часто состояла из инструментов с открытым исходным кодом (например, Nagios и Zenoss), измененных под заказчика и использованных в огромных масштабах: этого сложно было бы достичь с помощью лицензированных коммерческих программ того времени. Архитектура состоит из следующих компонентов.

- **Сбор данных на уровнях бизнес-логики, приложения и среды.** На каждом из этих уровней мы создаем телеметрию в виде событий, логов и показателей. Логи могут храниться в специфических для приложения файлах на каждом сервере (например, `/var/log/httpd-error.log`), но будет лучше, если все логи будут отсыльаться в одно и то же место, что упростит централизацию, ротацию и удаление. Такая услуга обеспечивается большинством операционных систем, например `syslog` для Linux, `Event Log` («Журнал событий») для Windows и так далее. Кроме того, мы собираем показатели на всех уровнях стека приложения, чтобы лучше понять, как ведет себя наша система. На уровне операционной системы можно собрать телеметрию работы центрального процессора, памяти, диска или использования сети с помощью таких инструментов, как `collectd`, `Ganglia` и так далее. Среди других собирающих данные инструментов можно назвать `AppDynamics`, `New Relic` и `Pingdom`.
- **Маршрутизатор событий, ответственный за хранение событий и показателей.** Этот компонент отвечает за визуализацию, отслеживание трендов, оповещение, обнаружение аномалий и многое другое. Собирая, храня и агрегируя нашу телеметрию, мы упрощаем последующий анализ и проверки работоспособности. Здесь же мы храним конфигурации, связанные с нашими службами (а также со связанными с ними приложениями и средой), и где, вероятно, оповещаем о превышении пороговых значений и проводим проверки работоспособности.

Собрав логи в одном месте, мы можем преобразовать их в показатели, подсчитывая их в маршрутизаторе событий; например, такое событие, как «`child pid 14024 exit signal Segmentation fault`», может быть подсчитано и просуммировано как один из показателей ошибки сегментации во всей производственной инфраструктуре.

Преобразовав логи в показатели, мы можем применить к ним статистические методы, например обнаружение отклонений от нормального состояния, чтобы выявлять выбросы и отклонения еще раньше. Скажем, настроить систему оповещения так, чтобы она сообщала о переходе от десяти ошибок сегментации на прошлой неделе к тысячам таких ошибок за последний час, что, конечно, требует тщательного расследования.

В дополнение к сбору телеметрии из служб и окружения необходимо также фиксировать важные события в процессе развертывания, например, когда программа проходит или проваливает автоматизированные тесты или когда мы развертываем продукт в любой среде. Кроме того, надо собирать данные, сколько времени требуется на выполнение сборок и на прохождение тестов. Так мы можем заметить условия, свидетельствующие о возможных проблемах, например если тест качества работы или сборка занимают в два раза больше времени, чем нужно. Это позволит найти и исправить ошибки до того, как они перейдут в фазу эксплуатации.



Рис. 26. Механизм мониторинга (источник: Торнбулл. The Art of Monitoring, Kindle edition, глава 2)

Далее нужно убедиться, что в нашу телеметрическую инфраструктуру легко заносить новую информацию и извлекать старую. Хорошо, если все будет организовано с помощью самообслуживающихся API, чтобы не приходилось оформлять тикет и долго ждать отчета.

В идеале нужно создать телеметрическую систему, информирующую нас, когда, а также где и как именно произошло нечто интересное. Эта система должна быть удобна для анализа вручную и автоматизированно, а также данные должны быть доступны для анализа без доступа к самому приложению, предоставившему логи. Как заметил Адриан Коккрофт, «Мониторинг настолько важен, что наши системы мониторинга должны быть более доступными и более гибкими, чем то, за чьим наблюдают».

Термин «телеметрия» синонимичен слову «показатели»: это все логи событий и все индикаторы служб на всех уровнях стека приложения, полученные как на этапе эксплуатации, так и до него, а также в процессе развертывания продукта.

Теперь, когда у нас есть инфраструктура централизованной телеметрии, мы должны удостовериться в том, что все разрабатываемые и используемые приложения поставляют достаточное количество нужных данных. Для этого инженеры разработки и эксплуатации должны генерировать телеметрию эксплуатации в процессе своей повседневной работы как для новых, так и для уже существующих служб.

Скотт Праф, главный архитектор и вице-президент отдела разработки компании CSG, отметил: «Каждый раз, когда NASA запускает ракету, тысячи автоматизированных сенсоров в ней отчитываются о состоянии каждого компонента этого ценнейшего объекта. А мы же часто не принимаем таких же мер предосторожности с нашими программами. Мы обнаружили, что создание телеметрии на уровнях приложения и инфраструктуры — одна из самых выгодных инвестиций, что мы когда-либо делали. В 2014 г. мы фиксировали более миллиарда событий в день, причем данные поступали из более чем сотни тысяч мест в коде».

В создаваемых и используемых нами приложениях каждый элемент функциональности должен находиться под наблюдением: если он был достаточно важен, чтобы его реализовать, значит, он достаточно важен, чтобы собирать о нем данные. Так мы сможем контролировать, что он работает именно так, как и предполагалось, и что итог работы такой, какого мы и хотели добиться.

Каждый участник процесса разработки и эксплуатации будет использовать телеметрию множеством разных способов. Например, разработчики могут временно увеличить количество собираемых данных, чтобы лучше определить проблему, а инженеры по эксплуатации будут опираться на телеметрию, чтобы лучше справляться с проблемами на стадии эксплуатации. Кроме

того, служба информационной безопасности и аудиторы могут анализировать сведения о работе приложения, чтобы убедиться в эффективности требуемого контроля, а менеджер по продукции будет использовать их для отслеживания бизнес-показателей, статистики использования функций программы или коэффициента конверсии.

Чтобы поддержать разнообразные модели использования, мы предлагаем разные уровни логирования. В них также можно настроить оповещения. Уровни могут быть такие.

- **Уровень отладки.** На этом уровне собирается информация обо всем, что происходит внутри приложения. Чаще всего этот уровень используется при соответственно отладке. Часто на стадии эксплуатации логи отладки отключают, но временно возвращаются к ним, если возникли какие-то проблемы.
- **Информационный уровень.** На этом уровне данные состоят из специфических для конкретной системы действий или же действий, совершаемых пользователем (например, «начало транзакции с использованием кредитной карты»).
- **Уровень предупреждений.** Здесь телеметрия сообщает нам о состояниях, потенциально порождающих проблемы (например, обращение к базе данных занимает больше времени, чем заранее запланировано). Эти условия, вероятно, выдадут оповещение и потребуют выявить и устранить неполадку, тогда как другие сообщения логов помогут нам лучше понять, какие действия привели к такому состоянию.
- **Уровень ошибок.** На этом уровне собирается информация об ошибках (например, падение при API-вызове или внутренняя ошибка).
- **Критический уровень.** Данные на этом уровне сообщают нам, когда мы должны прервать работу (например, сетевой агент (так называемый демон) не может подключиться к сетевому соединителю («сокету»)).

Выбор правильного уровня логирования важен. Дэн Норт, бывший консультант компании ThoughtWorks, принимавший участие в нескольких проектах, где были сформированы основные принципы непрерывной поставки ПО, замечает: «Когда вы решаете, должно ли сообщение звучать как ОШИБКА или ПРЕДУПРЕЖДЕНИЕ, представьте, что вас разбудили в четыре утра. Закончился тонер в принтере — это не ОШИБКА».

Чтобы убедиться, что у нас есть вся относящаяся к устойчивой и надежной работе приложения информация, нужно удостовериться, что все потенциально значимые события генерируют логи. Обязательно нужно учсть группы событий, собранные в списке Антона Чувакина, вице-президента по исследованиям, работающего в группе безопасности и риск-менеджмента подразделения GTP (Gartner for Technical Professionals) компании Gartner:

- решения о подтверждении прав доступа/авторизации (включая выход из системы);
- доступ в систему и доступ к данным;
- изменения системы и приложения (особенно изменения, связанные с правами доступа);
- изменения данных, такие как добавление, правка и удаление данных;
- некорректный ввод (возможный ввод вредоносных данных, угрозы и так далее);
- ресурсы (RAM, диск, процессор, пропускная способность соединения и любые другие ресурсы, имеющие жесткие или мягкие ограничения);
- работоспособность и доступность;
- загрузка и завершение работы;
- сбои и ошибки;
- срабатывание автоматического выключателя;
- задержки;
- успешное или неудачное резервное копирование.

Чтобы все эти логи было проще интерпретировать и понимать, стоит создать иерархические категории, такие как нефункциональные характеристики (например, качество работы, безопасность) и характеристики, связанные с функциональностью программы (например, поиск, ранжирование).

Как было описано в начале этой главы, высокопроизводительные компании используют дисциплинированный подход к решению проблем. Такой подход противоположен более распространенной практике использования слухов и домыслов, приводящей к столь печальному показателю, как количество среднего времени до признания невиновным: как быстро мы сможем убедить всех, что это не мы были причиной сбоя или простоя в работе.

Когда вокруг сбоев и проблем создана культура обвинений, команды могут не документировать изменения и скрывать показатели: ведь все могут увидеть, что они стараются избежать вины за возникновение проблем.

Другие негативные последствия отсутствия открытой телеметрии — напряженная атмосфера, необходимость защищаться от обвинений и, что хуже всего, неспособность получать общедоступные знания, как возникают проблемы и что необходимо, чтобы предотвратить их в будущем.

Телеметрия же позволяет нам использовать научный метод, чтобы формулировать гипотезы о причинах проблемы и о средствах ее устранения. Ниже следуют примеры вопросов, на которые можно ответить во время исправления ошибок и корректирования сбоев.

- Каковы доказательства того, что проблема действительно существует?
- Какие значимые события и изменения в наших приложениях и окружении могли привести к этой проблеме?
- Какие гипотезы мы можем сформулировать, чтобы подтвердить связь между предложенными причинами и следствиями?
- Как мы можем доказать, какие из гипотез верны и ведут к решению проблемы?

Ценность основанного на фактах решения проблем заключается не только в гораздо меньшем MTTR (и в лучших результатах для клиентов), но и в усилении взаимовыгодного сотрудничества между разработкой и ИТ-эксплуатацией.

Чтобы все могли выявлять и исправлять проблемы в процессе своей ежедневной работы, нужно выработать показатели, которые легко собирать, визуализировать и анализировать. Для этого мы должны создать такую инфраструктуру и библиотеки, чтобы всем и в разработке, и в эксплуатации было как можно проще придумывать и генерировать телеметрию для любой функциональности. Ее созданием они занимаются. В идеале это должна быть одна строка кода, создающая новый показатель и выводящая его на общий экран индикаторов, где его могут видеть все участвующие в процессе создания программы.

Эта философия привела к разработке одной из самых используемых библиотек показателей StatsD. Она появилась на свет в компании Etsy. Как сказал Джон Оллспоу, «Мы создали StatsD для того, чтобы разработчики больше не говорили: “Добавлять в мой код средства измерения слишком напряжно”. Теперь они могут делать это одной строчкой кода. Нам было важно, чтобы разработчики не считали добавление телеметрии такой же сложной задачей, как изменение схемы базы данных».

Программа StatsD может генерировать таймеры и счетчики с помощью одной строки кода (на Ruby, Perl, Python, Java и других языках), и ее часто используют вместе с Graphite или Grafana, преобразующих события в графики и панели индикаторов.

На рис. 27 приведен пример того, как одна строчка кода создает событие «вход пользователя в систему» (в этом случае строка на PHP: “StatsD:: increment(“login.successes”)). Итоговый график показывает количество успешных и неудачных входов за минуту, а вертикальные линии обозначают моменты, когда происходило развертывание продукта.

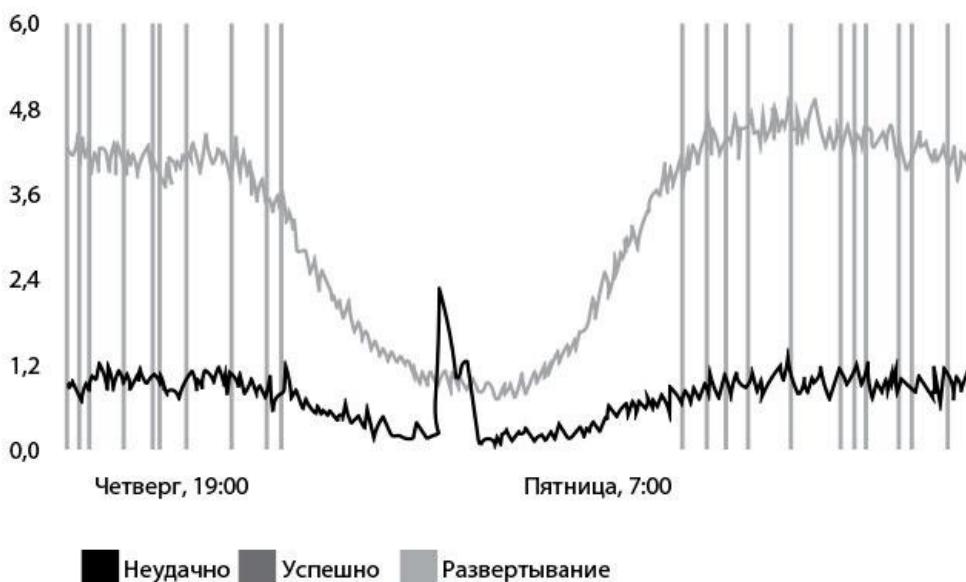


Рис. 27. Одна строка кода для генерирования телеметрии с помощью StatsD и Graphite в компании Etsy (источник: Йен Маллпасс, Measure Anything, Measure Everything)

Когда мы будем создавать графики по нашей телеметрии, мы также будем накладывать на них значимые события, возникающие в процессе эксплуатации, потому что нам известно, что большинство проблем на стадии эксплуатации происходит именно из-за эксплуатационных изменений, включая развертывание кода. Это часть того, что позволяет нам сохранять высокие темпы изменений и в то же время поддерживать высокий уровень безопасности системы.

Среди альтернативных StatsD библиотек, позволяющих разработчикам генерировать легко агрегируемую и анализируемую эксплуатационную телеметрию, можно назвать JMX и codahale metrics. Другие инструменты для сбора показателей — New Relic, AppDynamics и Dynatrace. Для того чтобы добиться схожей функциональности, можно также использовать munin и collectd.

### **Сделайте доступ к телеметрии свободным и создайте распространители информации**

На предыдущих шагах мы рассмотрели, как отделы эксплуатации могут создавать и улучшать эксплуатационную телеметрию в процессе ежедневной работы. На этом шаге нашей целью будет распространение этих сведений по всей организации. Нужно сделать так, чтобы любой сотрудник, нуждающийся в информации о любой службе или программе компании, мог получить ее без доступа к системе эксплуатации, без аккаунта со специальными правами доступа, без тикетов и без многодневного ожидания, пока кто-нибудь наконец построит нужный график.

Если телеметрия будет быстрой, легкодоступной и достаточно централизованной, у всех сотрудников будет одно и то же представление о реальном положении дел. Обычно это значит, что производственная телеметрия отображается на веб-страницах, сгенерированных централизованным сервером, например Graphite или любым другим, описанным в предыдущем разделе.

Мы хотим, чтобы телеметрия была очень заметной. Ее следует размещать там, где трудятся разработчики и работники служб эксплуатации. Так все заинтересованные смогут наблюдать за работой наших служб. Как минимум в число таких сотрудников нужно включить отдел разработки, ИТ-эксплуатации, менеджеров по продукции и службу информационной безопасности.

Такая система часто называется *распространителем информации*. Он определяется сообществом Agile Alliance как «общий для любого количества написанных от руки, нарисованных, напечатанных или электронных экранов, панелей, карт и мониторов, размещенных в визуально заметных местах таким образом, что любой сотрудник или прохожий может увидеть самую свежую информацию о количестве автоматизированных тестов, скорости, отчетах об ошибках, статусе непрерывной интеграции и так далее. Эта идея появилась как часть системы производства Тойота».

Размещая информационные распространители на видных местах, мы поощряем ответственность в нашей команде, активно воплощая следующие ценности:

- команде нечего прятать от посетителей (клиентов, стейкхолдеров и так далее);

- команде нечего прятать от себя: она признает проблемы и смотрит им в лицо.

Теперь, когда у нас есть инфраструктура для того, чтобы создавать и распространять телеметрию по всей организации, мы можем предоставлять эти данные как нашим внутренним клиентам, так и внешним. Например, можно вывесить доступные для широкой публики интернет-страницы. На них клиенты смогут узнавать, как работают необходимые им службы.

Эрнест Мюллер считает, что установление высокой степени прозрачности важно, хотя и встречает некоторое сопротивление:

«Одно из моих первых действий в новой организации — это использование распространителей информации для сообщения о проблемах и уточнения текущих изменений. Обычно это принимается подразделениями на ура. До этого они зачастую просто оставались в неведении. А для разработки и ИТ-эксплуатации, которые должны осуществляться вместе, чтобы получился хороший продукт, нужно постоянное взаимодействие, информирование и обратная связь».

Можно распространить эту прозрачность еще шире — вместо того чтобы оставлять проблемы, влияющие на клиентов, в секрете, мы можем передавать подобные сведения и нашим внешним клиентам. Это покажет, что мы ценим прозрачность, и тем самым поможет нам заслужить доверие заказчиков (см.).

Как уже было сказано в части III, компания LinkedIn появилась в 2003 г. Цель организации заключалась в том, чтобы ее пользователи благодаря своим связям «могли найти лучшие вакансии». К ноябрю 2015 г. у LinkedIn было более 350 миллионов участников, генерировавших десятки тысяч запросов в секунду, из-за чего на серверные системы компании приходилась нагрузка в миллионы запросов в секунду.

Прачи Гупта, технический директор организации LinkedIn, в 2011 г. написала о важности производственной телеметрии следующее: «В LinkedIn мы тщательно следим за тем, чтобы сайт был доступен и чтобы у наших пользователей в любой момент был доступ ко всей функциональности нашего сайта. Для выполнения этого обязательства необходимо обнаруживать сбои и разбираться с ними сразу, как только они возникают. Именно поэтому мы используем временные графики для наблюдения за сайтом, чтобы замечать инциденты и реагировать на них в течение нескольких минут... Такие методики мониторинга оказались отличным инструментом для инженеров. Они позволяют не терять темпа и дают время на обнаружение, приоритизацию и решение проблем».

Однако в 2010 г., несмотря на невероятно большие объемы генерирования телеметрии, инженерам было очень трудно получить доступ к данным, не говоря уже о том, чтобы проанализировать их. Так появилась на свет идея практики Эрика Вонга, проходившего летнюю стажировку в LinkedIn. Впоследствии она переросла в проект по созданию телеметрии, породивший инструмент InGraph.

Вонг писал: «Чтобы получить данные о чем-то простом, как, например, загрузка процессоров всех хостов, занятых каким-то конкретным процессом, нужно было заполнять тикет и ждать, пока что-нибудь потратит полчаса на составление отчета».

В то время для сбора телеметрии LinkedIn использовала Zenoss, но, как объясняет Вонг, «для получения данных из Zenoss надо было продираться сквозь медленный веб-интерфейс, поэтому, чтобы упростить процесс, я написал несколько скриптов на Python. Хотя там для установки сбора данных все равно требовалась настройка вручную, мне удалось уменьшить время на работу с интерфейсом Zenoss».

За лето Вонг продолжил наращивать функциональность InGraph, чтобы инженеры могли видеть то, что им нужно. Он добавил вычисления на нескольких наборах данных одновременно, еженедельные тренды для сравнения результативности и даже возможность создания заданных пользователем сводок, чтобы можно было выбирать, какие именно показатели будут отображаться на одной странице.

Говоря о результатах увеличенной функциональности InGraph и о ценности таких возможностей, Гупта отмечает: «Эффективность нашей системы мониторинга ярко проявилась в тот момент, когда у нас начали снижаться показатели, связанные с одним крупным веб-майл-провайдером, и этот провайдер обнаружил, что у них есть проблема в системе, только после того, как мы сообщили ему об этом!»

То, что началось как проект одной летней стажировки, теперь одна из самых заметных частей ИТ-эксплуатации LinkedIn. InGraph был настолько успешен, что теперь строящиеся в реальном времени

графики всегда на виду в технических офисах компании, где не заметить их просто невозможно.

К этому моменту мы создали инфраструктуру, необходимую для быстрого сбора телеметрии на всех уровнях стека приложения и ее распространения по всей организации.

На этом шаге мы найдем все пробелы в нашей телеметрии, мешающие нам быстро обнаруживать и устранять проблемы. Это особенно важно, если у разработки и ИТ-эксплуатации на нынешний момент очень мало данных (или их вообще нет). Мы потом используем эти сведения, чтобы лучше предсказывать возникновение проблем, а также чтобы все могли получать необходимую для принятия верных решений информацию.

Для достижения этой цели нам нужно создать достаточно телеметрии на всех уровнях стека приложений для всех окружений, а также для связанных с ними этапов непрерывного развертывания. Нам необходима телеметрия следующих уровней:

- **бизнес-уровень.** Примерами могут быть количество торговых сделок, доход от сделок, количество регистраций пользователей, коэффициент утечки клиентов, результаты А/В-тестирования и так далее;
- **уровень приложения.** Это, например, время проведения транзакций, время ответа пользователя, сбои в работе приложения и так далее;
- **уровень инфраструктуры (базы данных, операционная система, сетевые подключения, память).** Здесь примеры — трафик веб-сервера, загруженность процессора, использование диска и так далее;
- **уровень клиентского ПО (JavaScript клиента браузера, мобильное приложение).** Это, например, ошибки и падения приложения, время транзакций пользователя и так далее;
- **уровень развертывания.** Примерами могут быть статус сборки (красный или зеленый для разных наборов автоматизированных тестов), изменение необходимого на развертывание времени, частота развертываний, улучшение тестовой среды и статус среды.

Если у нас будет достаточная телеметрия во всех этих областях, мы сможем увидеть работоспособность всего того, от чего зависит наш продукт. Вместо слухов, домыслов и обвинений у нас окажутся факты и данные.

Далее, регистрируя все дефекты и сбои в приложениях и инфраструктуре (например, аварийные завершения работы, ошибки приложения, исключения, ошибки в работе запоминающих устройств и серверов), мы можем лучше фиксировать события, связанные с безопасностью. Такая телеметрия не только информирует разработку и эксплуатацию о падении программ и служб, но также сообщает о возможных уязвимых местах системы безопасности.

Раньше замечая и корректируя ошибки, мы можем разбираться с ними, пока они еще невелики и легко исправимы, а число клиентов, испытывающих последствия, мало. Кроме того, после каждого сбоя в производстве нужно определить, какой телеметрии не хватает, какие данные могли бы обеспечить более быстрое обнаружение и восстановление. Еще лучше, если бы мы могли заполнить эти пробелы на стадии разработки функциональности с помощью проверки нашей работы коллегами.

На уровне приложения наша цель — убедиться, что мы создаем телеметрию, не только отражающую работоспособность этого приложения (например, использование памяти, число транзакций и так далее), но также измеряющую, насколько мы достигаем целей нашей организации (например, число новых пользователей, количество входов в систему, длина пользовательской сессии, процент активных пользователей, частота использования тех или иных функциональных возможностей и тому подобное).

К примеру, если у нас есть сервис, поддерживающий электронную торговлю, нам нужно удостовериться, что у нас есть данные обо всех пользовательских событиях, ведущих к успешной транзакции, приносящей доход. Затем мы можем отслеживать все действия пользователя, необходимые для достижения желаемого результата.

Эти показатели будут разными в зависимости от сферы деятельности и целей компании. Например, для сайтов электронной торговли целью может быть максимизация времени, проведенного на сайте. Однако поисковым службам нужно ориентироваться на сокращение времени пребывания на странице, поскольку долгие сессии говорят о том, что пользователи не могут найти то, что хотят.

В общем случае бизнес-метрики будут частью *воронки приобретения клиентов*, то есть образного представления теоретических шагов, совершаемых потенциальным покупателем, решившим сделать покупку. Например, для сайта электронной торговли измеримыми событиями на этом пути будут общее время, проведенное на сайте, переходы по ссылкам на товары, добавление товаров в корзину и завершенные заказы.

Эд Бланкеншип, старший менеджер по продукции Microsoft Visual Studio Team Services, пишет: «Часто команды по разработке элемента функциональности определяют свои цели с помощью воронки приобретения клиентов. Их цель — сделать так, чтобы клиенты пользовались их функциональностью каждый день. Разные группы пользователей иногда неформально называют “зеваками”, “активными пользователями”, “вовлеченными пользователями” и “глубоко вовлеченными пользователями”. Для каждой такой стадии есть своя телеметрия».

Наша цель — сделать каждый бизнес-показатель действенным. Эти важнейшие индикаторы должны сообщать нам, как изменить продукт, и быть гибкими для экспериментирования и А/В-тестирования. Когда метрика не ведет к непосредственным действиям, скорее всего, это просто пустые индикаторы, предоставляющие мало полезной информации. Такие данные стоит хранить, но вот выставлять на обозрение не нужно и уж тем более не стоит бить из-за них тревогу.

В идеале любой, кто смотрит на наши распространители информации, должен суметь понять, как отображенные сведения связаны с целями компании, такими как доход, приобретение покупателей, коэффициент конверсии и так далее. Нужно определить и связать каждый показатель с бизнес-показателями на самых ранних стадиях определения функциональности и разработки и измерять результаты после развертывания в стадию эксплуатации. Кроме того, это помогает представителям заказчика описывать бизнес-контекст каждого элемента функциональности для всех в потоке создания ценности.

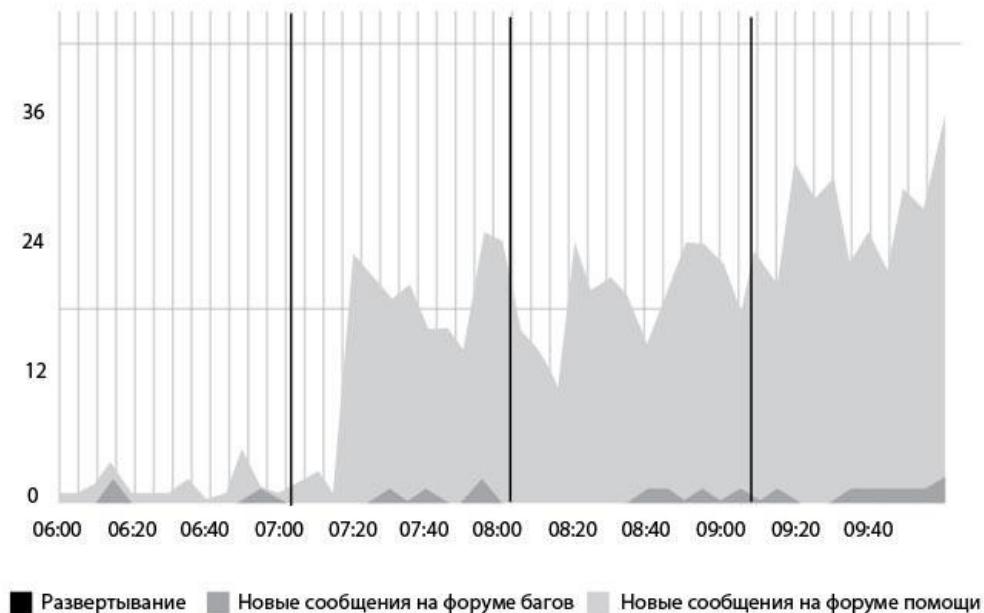


Рис. 28. Активность пользователей в связи с новой функциональностью после развертываний, измеренная в количестве форумных сообщений (источник: Майк Бриттен, “Tracking Every Release”, взято с сайта , 8 декабря 2010 г., )

Можно задать более общий бизнес-контекст, если иметь в виду и отображать на графиках периоды, важные с точки зрения ведения деятельности и высокого уровня бизнес-планирования. Примерами могут служить временные отрезки, на которые приходится большое число транзакций: пики продаж, связанные с праздниками, закрытие финансовых периодов в конце кварталов или запланированные аудиторские проверки. Эту информацию можно использовать как напоминание для того, чтобы не вносить существенные изменения в периоды, когда доступность сервиса крайне важна, или избегать каких-либо действий в разгар аудиторской проверки.

Распространяя информацию о действиях пользователей в контексте наших целей, мы получаем быструю обратную связь для команд, занятых конкретными элементами функциональности. Так они могут выяснить, действительно ли их сервисы используются и в какой степени они соответствуют бизнес-целям. В результате мы закрепляем ожидания, что мониторинг и анализ действий клиентов также часть нашей ежедневной работы, и сами лучше понимаем, как наша работа способствует достижению целей организации.

Точно так же, как и для показателей на уровне приложения, для индикаторов эксплуатационной и не эксплуатационной инфраструктуры наша цель — убедиться, что мы создаем достаточно телеметрии, чтобы быстро выяснить, действительно ли инфраструктура — одна из причин неполадки или нет. Кроме того, мы должны быть способны четко определить, что именно в инфраструктуре создает проблемы (например, базы данных, операционная система, запоминающее устройство, сетевые соединения и так далее).

Мы хотим сделать прозрачной как можно большую часть инфраструктурной телеметрии, по всем компаниям, чьи программы и продукты мы используем. В идеале она должна быть организована по сервисам или по приложениям. Другими словами, когда в нашем окружении что-то идет не так, мы должны точно знать, на какие приложения и сервисы это влияет или может потенциально повлиять.

Раньше связи между сервисом и эксплуатационной инфраструктурой, от которой он зависел, создавались вручную (например, База данных управления конфигурациями (CMDB), ITIL или создание определений конфигурации в инструментах оповещения, например в Nagios). Однако сейчас эти связи все чаще регистрируются в сервисах автоматически, затем они открываются в динамическом режиме и используются в эксплуатации с помощью таких инструментов, как Zookeeper, Etcd, Consul и так далее.

Эти инструменты позволяют сервисам регистрироваться самостоятельно, сохраняя информацию, необходимую для других сервисов (например, IP-адрес, номера портов, URI). Такой подход решает проблему управления вручную базой данных конфигураций ITIL. Это особенно необходимо, когда сервисы состоят из сотен (а иногда тысяч и даже миллионов) узлов, каждый с динамически присваиваемым IP-адресом.

Вне зависимости от того, насколько просты или сложны наши службы, составление графиков по бизнес-метрикам и показателям инфраструктуры одновременно позволяет отслеживать разные проблемы. Например, мы можем увидеть, что регистрация новых пользователей упала на 20 % по сравнению с ежедневной нормой и в то же время все запросы к базам данных стали проводиться в пять раз дольше. Причины проблемы стали более ясными.

Кроме того, бизнес-метрика создает контекст для индикаторов инфраструктуры, благодаря чему облегчается совместная работа разработки и эксплуатации. По наблюдениям Джоди Малки, СТО компании Ticketmaster/LiveNation, «вместо того чтобы оценивать отдел эксплуатации относительно потерянного времени, я считаю, что гораздо лучше оценивать разработку и эксплуатацию относительно реальных бизнес-последствий потерянного времени: какой доход мы должны были получить, но не смогли».

Отметим, что вдобавок к мониторингу служб эксплуатации нам также необходима телеметрия этих же служб в доэксплуатационном окружении (например, окружение разработчика, тестовое окружение, окружение, эмулирующее реальную среду, и так далее). Это позволяет нам обнаружить и устраниć проблемы до того, как они перейдут в эксплуатацию, например такую неприятность, как все увеличивающееся время на внесение новых сведений в базу данных из-за отсутствующего индекса таблицы.

Даже после того как мы создали систему непрерывного развертывания, позволяющую вносить частые и небольшие изменения, все равно остается риск. Эксплуатационные побочные эффекты — это не только сбои и простой в работе, но и значительные срывы и отклонения от привычного процесса сопровождения продукта.

Чтобы сделать изменения видимыми, мы накладываем всю информацию о развертывании и эксплуатации на графики. Например, для службы, управляющей большим количеством входящих транзакций, такие изменения могут приводить к долгому периоду с низкой работоспособностью, пока не восстановится система поиска в кэш-памяти.

Чтобы лучше понимать и сохранять качество служб, нужно понять, насколько быстро уровень работоспособности вернется в норму, и, если это необходимо, принять меры для улучшения этого уровня.

Нам также нужно визуализировать сведения о действиях по развертыванию и эксплуатации (например, о том, что проводится техобслуживание или резервное копирование) в тех местах, где мы хотим отображать оповещения или же, наоборот, отменить их вывод.

Польза эксплуатационной телеметрии от Etsy и LinkedIn показывает, насколько важно замечать проблемы сразу в момент появления, чтобы можно было найти и устранить их причину и быстро исправить ситуацию. Когда все элементы нашей службы, будь то приложение, база данных или окружение, создают легко анализируемую телеметрию, к которой есть удобный доступ, мы можем отловить неполадки задолго до того, как они приведут к катастрофическим последствиям. В идеале — задолго до того, как заказчик заметит, что что-то пошло не так. Результат — не только счастливые клиенты, но и благодаря отсутствию кризисов и авралов более благоприятная и

продуктивная рабочая атмосфера без стрессов и выгораний.

## **Глава 15. Анализируйте телеметрию, чтобы лучше предсказывать проблемы и добиваться поставленных целей**

Как мы видели в предыдущей главе, нам нужно достаточное количество телеметрии в наших приложениях и инфраструктуре, для того чтобы видеть и решать проблемы в момент появления. В этой главе мы создадим инструменты для обнаружения спрятанных в наших данных отклонений и слабых сигналов о неполадках, помогающих избежать катастрофических последствий. Мы опишем многочисленные статистические методики, а также покажем на реальных примерах, как они работают.

Отличный пример анализа телеметрии для проактивного поиска ошибок до того, как они повлияют на клиентов, — компания Netflix, крупнейший провайдер фильмов и сериалов на основе потокового мультимедиа. В 2015 г. доход Netflix от 75 миллионов подписчиков составлял 6,2 миллиарда долларов. Одна из ее целей — обеспечить наилучшие впечатления от просмотра видео онлайн во всех уголках земного шара. Для этого нужна устойчивая, масштабируемая и гибкая инфраструктура. Рой Рапопорт описывает одну из сложных задач управления облачной службой доставки видео: «Допустим, у вас есть стадо скота. В нем все должны выглядеть и вести себя одинаково. Какая корова отличается от остальных? Или, более конкретно, у нас есть вычислительный кластер с тысячью узлов. Они не фиксируют свое состояние, на них работает одно и то же программное обеспечение, и через них проходит один и тот же объем трафика. Задача в том, чтобы найти узлы, не похожие на остальные».

Одной из статистических методик, использованных в Netflix в 2012 г., было обнаружение выбросов. Виктория Ходж и Джим Остин из Йоркского университета определяют ее как обнаружение «аномальных рабочих состояний, приводящих к значительному ухудшению работоспособности, например дефект вращения двигателя самолета или нарушение потока в трубопроводе».

Рапопорт объясняет, что Netflix «обнаруживал выбросы очень простым способом. Сначала определялась отправная точка, то, что считалось нормальным прямо сейчас в этом наборе узлов вычислительного кластера. Затем искали узлы, выбивающиеся из общей картины, и удаляли их из эксплуатации».

Он продолжает: «Мы можем автоматически помечать неправильно ведущие себя узлы без того, чтобы выяснить, что же такое правильное поведение. И поскольку мы настроены на гибкую работу в облаке, инженеров эксплуатации мы ни о чем не просим. Мы просто отстреливаем больной или безобразничающий узел, затем составляем лог или сообщаем об этом инженерам так, как им удобно».

Применяя методику обнаружения выбросов среди серверов, Netflix, по словам Рапопорта, «существенно сократил усилия на выявление больных серверов и, что более важно, сильно уменьшил время на их починку. Результат — улучшение качества услуг. Выигрыш от этих методик для сохранения рассудка сотрудников, баланса работы и жизни и качества услуг переоценить сложно». Сделанное Netflix подчеркивает один очень конкретный способ использования телеметрии для устранения проблем, прежде чем они повлияют на клиентов.

В этой главе мы исследуем разные статистические и визуализирующие методики (включая обнаружение выбросов) для анализа телеметрии и предсказывания возможных проблем. Это позволит решать проблемы быстрее, дешевле и раньше, до того как их заметят клиенты или ваши коллеги. Кроме того, мы более полно опишем связь собранных данных и особенностей ситуации, что позволит нам принимать более эффективные решения и добиваться целей организации.

Один из простейших статистических способов анализа производственного показателя — это расчет среднего и стандартных отклонений. С их помощью мы можем создать фильтр, оповещающий, что показатель сильно отклоняется от обычных значений, и даже настроить систему оповещения, чтобы мы могли сразу предпринять нужные действия (например, если запросы в базе данных обрабатываются значительно медленнее, то в 2:00 система сообщит дежурным сотрудникам, что нужно провести расследование).

Когда в важнейших службах возникает проблема, будить людей в два часа ночи — правильно. Однако, когда мы создаем оповещения, не содержащие никакой программы разумных действий, или же когда происходит ложное срабатывание, исполнителей мы будим зря. Как отмечает Джон Винсент, стоявший у истоков движения DevOps, «переутомление от чрезмерного количества сигналов тревоги — наша самая главная проблема на этот момент... Нужно разумнее подходить к оповещениям, или мы все сойдем с ума».

Оповещения можно сделать лучше, если увеличить отношение сигнала к шуму, фокусируясь на значимых отклонениях или выбросах. Предположим, что нам надо проанализировать число несанкционированных входов в систему в день. У собранных данных — распределение Гаусса (то есть нормальное распределение), совпадающее с графиком на рис. 29. Вертикальная линия в

середине колоколообразной кривой — среднее, а первые, вторые и трети стандартные отклонения, обозначенные другими вертикальными линиями, содержат 68, 95 и 99,7 % наблюдений соответственно.

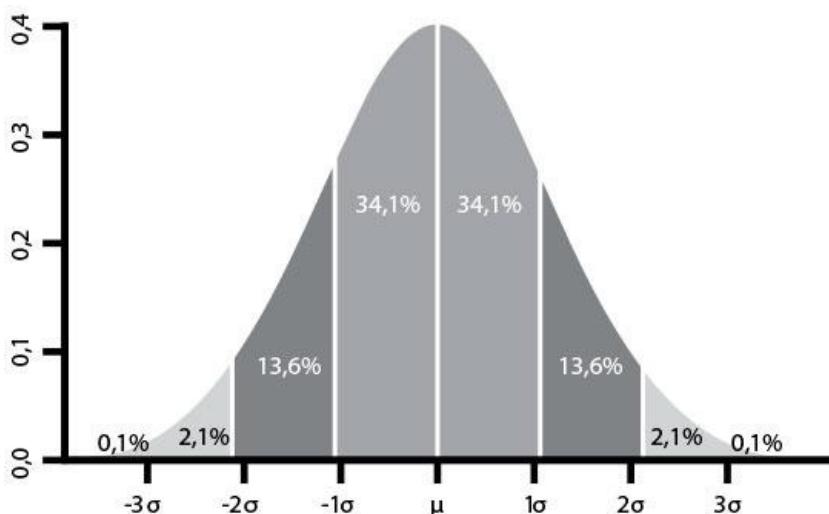


Рис. 29. Стандартные отклонения ( $\sigma$ ) и среднее ( $\mu$ ) распределения Гаусса (источник: «Википедия», статья Normal Distribution, )

Обычный способ использовать стандартные отклонения — время от времени проверять набор данных по какому-то показателю и сообщать, если он сильно отличается от среднего значения. Например, оповещение сработает, если число попыток несанкционированных входов в систему в день больше среднего на три величины стандартного отклонения. При условии, что данные распределены нормально, только 0,3 % всех событий будут включать сигнал тревоги.

Даже такая простая методика статистического анализа ценна, потому что никому не нужно определять пороговое значение. При отслеживании тысяч и сотен тысяч показателей это было бы невыполнимой задачей.

Далее в тексте термины *телеметрия, показатель и наборы данных* будут использоваться как синонимы. Другими словами, показатель (например, время загрузки страницы) будет увязываться с набором данных (например, 2 мс, 8 мс, 11 мс и так далее) Набор данных — статистический термин, обозначающий матрицу значений показателей, где каждый столбец представляет переменную, над которой производятся те или иные статистические операции.

Том Лимончелли, соавтор книги The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems и бывший инженер по обеспечению надежности сайтов компании Google, говорит о мониторинге показателей следующее: «Когда коллеги просят меня объяснить, что именно нужно мониторить, я шучу, что в идеальном мире мы удалили бы все имеющиеся оповещения в нашей системе наблюдения. Затем после каждого сбоя мы спрашивали бы себя: какие индикаторы могли бы предсказать этот сбой? И добавляли бы эти индикаторы в систему, настраивая соответствующие оповещения. И так снова и снова. В итоге у нас были бы только оповещения, предотвращающие сбои, тогда как обычно нас заваливают сигналами тревоги уже после того, как что-то сломалось».

На этом шаге мы воспроизведем результаты такого упражнения. Один из простейших способов добиться этого — это проанализировать самые серьезные инциденты за недавнее время (например, 30 дней) и создать список телеметрии, делающей возможной более раннюю и быструю фиксацию и диагностику проблемы, а также легкое и быстрое подтверждение того, что лекарство применено успешно.

Например, если наш веб-сервер NGINX перестал отвечать на запросы, мы могли бы взглянуть на индикаторы: они заблаговременно предупредили бы нас — что-то идет не так. Такими показателями могут быть:

- **уровень приложения:** увеличившееся время загрузки веб-страниц и так далее;
- **уровень ОС:** низкий уровень свободной памяти сервера, заканчивающееся место на диске и так далее;

- **уровень баз данных:** более долгие транзакции баз данных и так далее;
- **уровень сети:** упавшее число функционирующих серверов на балансировщике нагрузки и так далее.

Каждый из этих показателей — потенциальный предвестник аварии. Для каждого мы могли бы настроить систему оповещения, если они будут сильно отклоняться от среднего значения, чтобы мы могли принять меры.

Повторяя этот процесс для все более слабых сигналов, мы будем обнаруживать проблемы все раньше, и в результате ошибки будут затрагивать все меньше клиентов. Другими словами, мы и предотвращаем проблемы, и быстрее их замечаем и устраняем.

### **Проблемы телеметрии, имеющей негауссовое распределение**

Использование средних и стандартных отклонений для фиксации выбросов может быть очень полезным. Однако на некоторых наборах данных, используемых в эксплуатации, эти методики не будут давать желаемых результатов. Как отмечает Туфик Бубе, «нас будут будить не только в два часа ночи, но и в 2:37, 4:13, 5:17. Это происходит, когда у наших данных не нормальное распределение».

Другими словами, когда плотность распределения наблюдений описывается не показанной выше колоколообразной кривой, привычные свойства стандартных отклонений использовать нельзя. Например, представим, что мы наблюдаем за количеством скачиваний файла с нашего сайта в минуту. Нам нужно определить периоды, когда у нас необычно высокое число скачиваний. Пусть это число будет больше, чем три стандартных отклонения от среднего. Тогда мы сможем заранее увеличивать мощность или пропускную способность.

Рис. 30 показывает число одновременных скачиваний в минуту. Когда участок линии сверху графика выделен черным цветом, количество скачиваний в заданный период (иногда называемый «скользящим окном») превышает заданную величину. В противном случае линия окрашена в серый цвет.

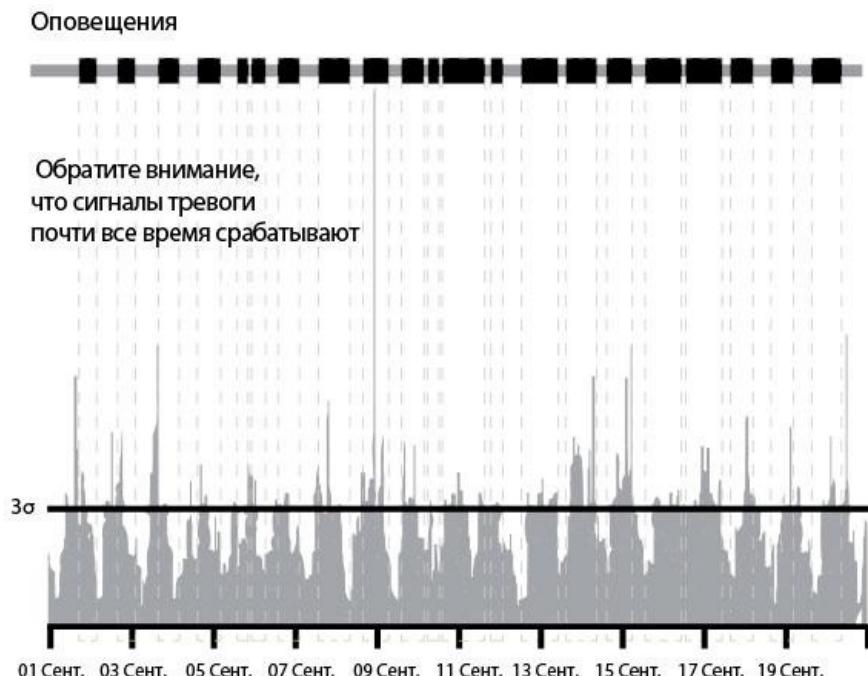


Рис. 30. Число загрузок в минуту: чрезмерное оповещение о проблемах при использовании правила трех стандартных отклонений (источник: Туфик Бубе, “Simple math for anomaly detection”)

График наглядно показывает очевидную проблему: оповещения идут практически непрерывным потоком. Это происходит потому, что почти в любой период у нас есть моменты, когда число скачиваний превышает порог в три стандартных отклонения.

Чтобы доказать это, построим гистограмму (рис. 31). На ней показана частота скачиваний в минуту.

Форма гистограммы отличается от классической куполообразной кривой. Вместо этого распределение явно скошено к левому краю. Это говорит нам о том, что большую часть времени у нас малое число скачиваний в минуту, но при этом число скачиваний очень часто превышает предел в три стандартных отклонения.



Рис. 31. Число скачиваний в минуту: гистограмма данных, имеющих негауссово распределение (источник: Туфик Бубе, "Simple math for anomaly detection")

У многих реальных наборов данных распределение не нормально. Николь Форсгрен объясняет: «В эксплуатации у многих наших комплектов данных так называемое распределение хи-квадрат. Использование стандартных отклонений для них не только приводит к чрезмерному или недостаточному количеству оповещений, но и просто выдает бесмысленные результаты». Далее, Николь отмечает: «Когда вы считаете число одновременных скачиваний, которое на три стандартных отклонения меньше среднего, вы получаете отрицательное число. А это явно бесмысленно».

Чрезмерное количество оповещений приводит к тому, что инженеров эксплуатации часто будят среди ночи и долго держат на ногах, даже когда они мало что могут сделать. Проблема недостаточного оповещения о проблемах также весьма значительна. Например, предположим, что мы наблюдаем число завершенных транзакций и из-за отказа какого-то компонента количество транзакций в середине дня внезапно падает на 50 %. Если эта величина находится в пределах трех стандартных отклонений от среднего, никакого сигнала тревоги подано не будет, а значит, наши клиенты обнаружат эту проблему раньше, чем мы. Если дойдет до этого, решить проблему будет гораздо сложнее.

К счастью, для выявления аномалий в наборах данных, имеющих не нормальное распределение, тоже есть специальные методики. О них мы расскажем ниже.

Еще один инструмент, разработанный в Netflix для улучшения качества услуг, Scryer, борется с некоторыми недостатками сервиса Auto Scaling компании Amazon (далее — AAS), который динамически увеличивает и уменьшает количество вычислительных серверов AWS в зависимости от данных по нагрузке. Система Scryer на основе прошлого поведения пользователя предсказывает, что именно ему может потребоваться, и предоставляет нужные ресурсы.

Scryer решил три проблемы AAS. Первая заключалась в обработке резких пиков нагрузки. Поскольку время включения инстансов AWS составляет от 10 до 45 минут, дополнительные вычислительные мощности часто загружались слишком поздно, чтобы справиться с пиковыми нагрузками. Суть второй проблемы в следующем: после сбоев быстрый спад пользовательского спроса приводил к тому, что AAS отключал слишком много вычислительных мощностей и потом неправлялся с последующим увеличением нагрузки. Третья проблема — AAS не учитывал в расчетах известные ему паттерны использования трафика.

Netflix воспользовался тем фактом, что паттерны поведения его пользователей были на удивление устойчивыми и предсказуемыми, несмотря на то что их распределение не было нормальным. На графике выше отображено количество пользовательских запросов в секунду на протяжении

рабочей недели. Схема активности клиентов с понедельника по пятницу постоянна и устойчива.

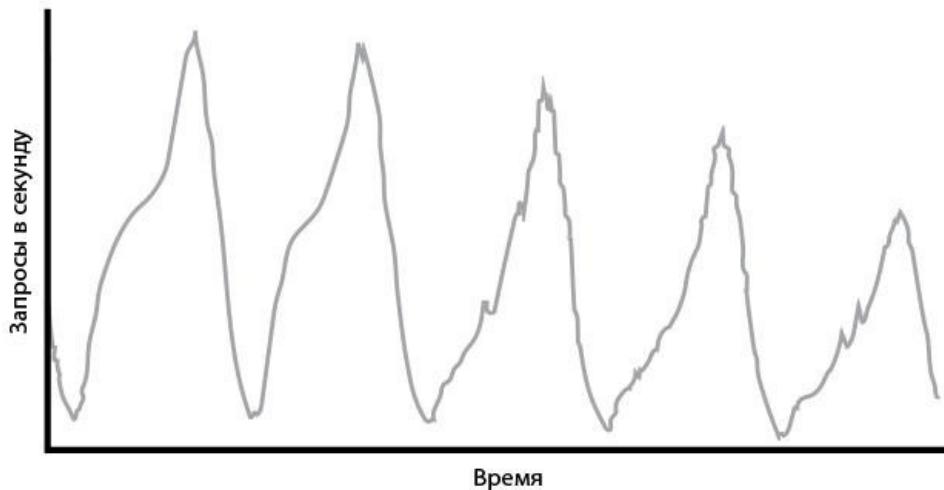


Рис. 32. Пользовательский спрос в течение пяти дней, по данным компании Netflix (источник: Дэниэл Джейкобсон, Дэнни Юан и Нирадж Джоши, "Scryer: Netflix's Predictive Auto Scaling Engine," The Netflix Tech Blog, 5 ноября 2013 г., )

Scryer использует комбинацию методик выявления выбросов, чтобы избавляться от сомнительных наблюдений, и затем для сглаживания данных применяет такие методики, как быстрое преобразование Фурье или линейная регрессия, в то же время сохраняя структуру повторяющихся пиков. В результате Netflix может предсказывать требуемый объем трафика с удивительной точностью.

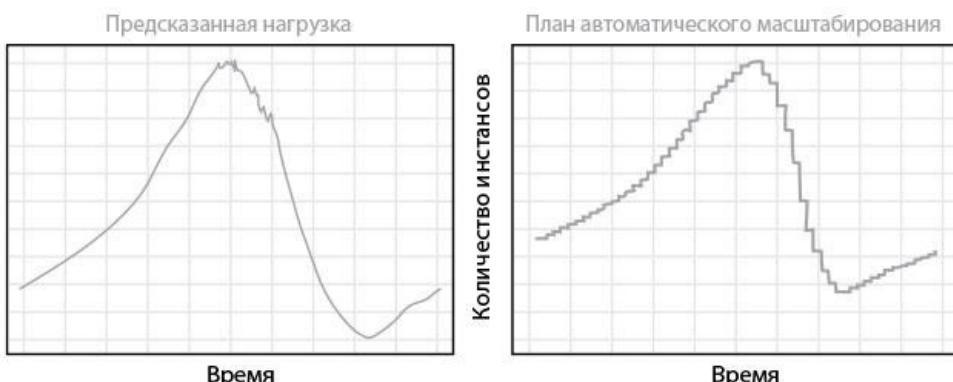


Рис. 33. Предсказание пользовательского трафика системой Scryer компании Netflix и итоговый план распределения вычислительных ресурсов AWS (источник: Джейкобсон, Юан, Джоши, "Scryer: Netflix's Predictive Auto Scaling Engine")

Всего лишь за несколько месяцев после введения в эксплуатацию системы Scryer Netflix значительно улучшил качество обслуживания клиентов и доступность своих сервисов, а также уменьшил расходы на использование Amazon EC2 (сервис облачного хостинга).

Когда у наших данных распределение не нормально, мы все равно можем обнаружить значимые отклонения с помощью разных методов. Здесь пригодится обширный класс методик выявления аномалий, часто определяемых как «поиск явлений и событий, не подчиняющихся ожидаемым правилам и моделям». Некоторые подобные возможности имеются среди наших инструментов мониторинга, для использования других может потребоваться помочь тех, кто разбирается в статистике.

Тарун Редди, вице-президент по разработке и эксплуатации компании Rally Software, выступает за активное сотрудничество между эксплуатацией и статистикой: «Чтобы поддерживать качество наших услуг, мы помещаем все наши производственные показатели в Tableau, программный пакет статистического анализа. У нас даже есть обученный статистике инженер эксплуатации, пишущий код на R (еще один статистический пакет). У этого инженера есть свой журнал запросов, заполненный просьбами других команд посчитать отклонения как можно раньше, прежде чем они приведут к более значительным отклонениям, которые могли бы повлиять на заказчиков».

Одна из пригодных статистических методик — это сглаживание. Оно особенно хорошо подходит, если данные — временные ряды: если у каждого наблюдения есть отметка о времени (например, скачивание файла, завершенная транзакция и так далее). Сглаживание часто используется

скользящее среднее, преобразующее наши данные с помощью усреднения каждого наблюдения с соседними наблюдениями в пределах заданного «окна». Это сглаживает кратковременные колебания и подчеркивает долговременные тренды или циклы.

Пример сглаживающего эффекта показан на рис. 34. Синяя линия показывает необработанные данные, тогда как черная отображает тридцатидневное скользящее среднее (то есть среднее последних тридцати дней).

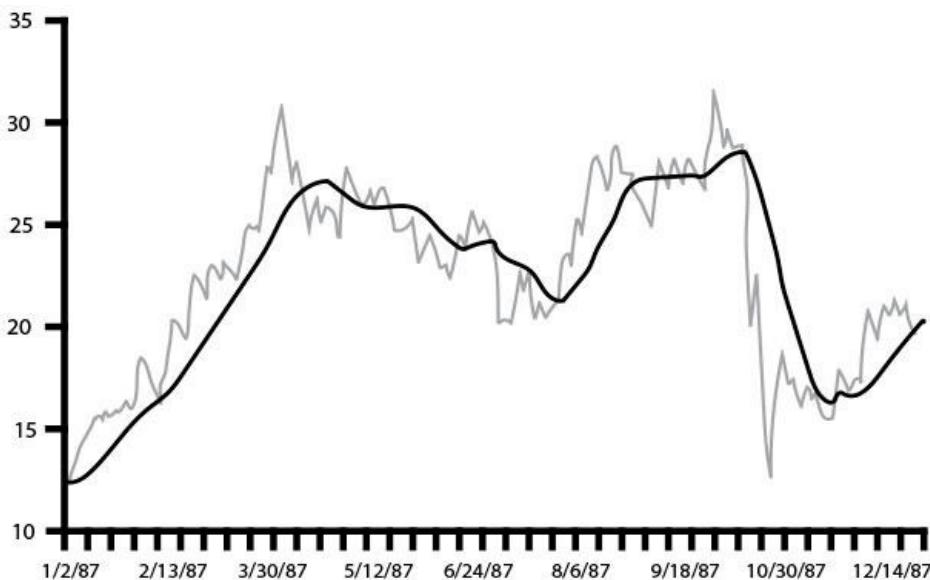


Рис. 34. Цена акции компании Autodesk и сглаживание с помощью тридцатидневного скользящего среднего (источник: Джейкобсон, Юан, Джоши, "Scryer: Netflix's Predictive Auto Scaling Engine")

Существуют и более экзотические методики обработки данных, например быстрое преобразование Фурье. Оно широко использовалось в обработке изображений. Или тест Колмогорова — Смирнова (он включен в Graphite и Grafana), часто используемый, чтобы найти сходства или различия в периодических или сезонных данных.

Можно с уверенностью ожидать, что большой процент нашей телеметрии о поведении пользователей будет иметь периодический или сезонный характер: веб-трафик, покупки, просмотр фильмов и многие другие действия клиентов регулярны и имеют на удивление предсказуемые ежедневные, недельные или годовые паттерны. Благодаря этому мы можем фиксировать отклоняющиеся от привычных поведенческих схем ситуации, например если уровень транзакций, связанных с заказами, вечером во вторник вдруг упадет на 50 % по сравнению с обычным значением.

Эти методики очень цепны для построения прогнозов, поэтому можно попробовать поискать в отделах маркетинга или бизнес-аналитики специалистов, обладающих нужными для анализа знаниями и навыками. Возможно, с такими сотрудниками мы сможем определить общие проблемы и использовать улучшенные методики выявления аномалий и предсказания ошибок для достижения общих целей.

На конференции Monitorama в 2014 г. Туфик Бубе поведал о мощности методик по выявлению аномалий, особо подчеркивая эффективность теста Колмогорова — Смирнова. Его часто используют в статистике для определения значимости различий между двумя наборами данных (его можно найти в таких популярных инструментах, как Graphite и Grafana). Цель разбора этого практического примера — не показать алгоритм решения какой-то проблемы, а продемонстрировать, как класс определенных статистических методик можно использовать в работе и как он используется в нашей организации в совершенно разных приложениях.

На рис 35 показано количество транзакций в минуту на сайте электронной торговли. Обратите внимание на недельную периодичность графика: число транзакций падает в выходные. Просто взглянув на этот график, можно заметить, что на четвертой неделе произошло что-то необычное: количество транзакций в понедельник не вернулось к обычному уровню. Это происшествие стоит исследовать.

## Оповещения

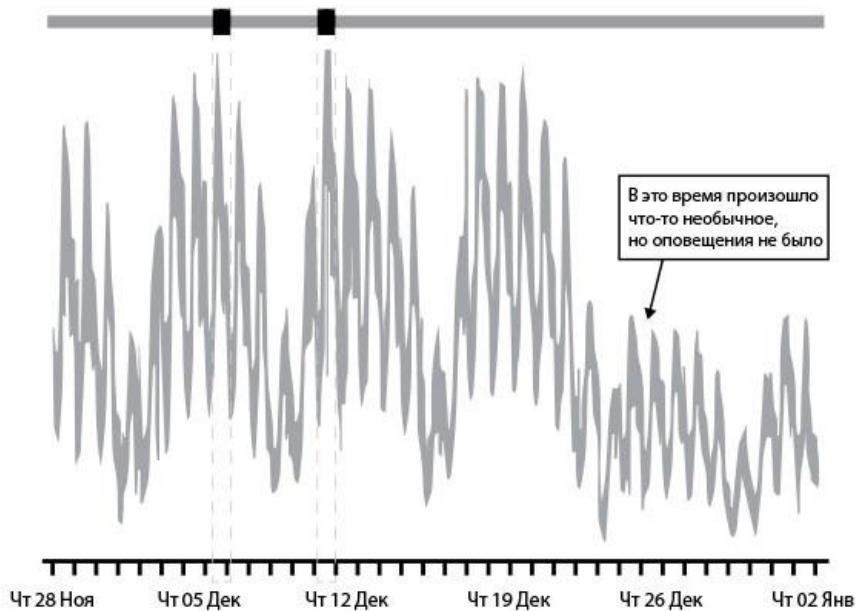


Рис. 35. Количество транзакций: отсутствие нужного оповещения при использовании правила трех отклонений (источник: Туфик Бубе, "Simple math for anomaly detection")

Использование правила трех стандартных отклонений выдало бы оповещения только дважды и пропустило бы важное падение количества транзакций в понедельник четвертой недели. В идеале мы хотели бы получить оповещение и о том, что данные отклонились от привычного паттерна.

«Даже просто сказать слова “Колмогоров — Смирнов” — отличный способ всех впечатлить, — шутит Бубе. — Но вот о чем инженеры эксплуатации должны сказать специалистам по статистике, так это то, что эти непараметрические критерии отлично подходят для данных со стадии эксплуатации, потому что не нужно делать никаких допущений о нормальности или других свойствах распределения. А ведь крайне важно понять, что же происходит в наших сложных системах. Пользуясь методиками, мы сравниваем два вероятностных распределения, в данном случае периодические и сезонные данные. Это помогает нам найти отклонения в ежедневно или еженедельно меняющихся данных».

На рис. 36 показан тот же самый набор данных, но с примененным фильтром Колмогорова — Смирнова. Третий выделенный черным участок соотносится с аномальным понедельником, когда уровень транзакций не вернулся к обычному уровню. Применение этой методики оповещает о проблемах в системе, практически не видных с помощью простого визуального осмотра или стандартных отклонений. При таком подходе раннее обнаружение не позволит проблеме повлиять на клиентов, а также поможет реализовать цели организации.

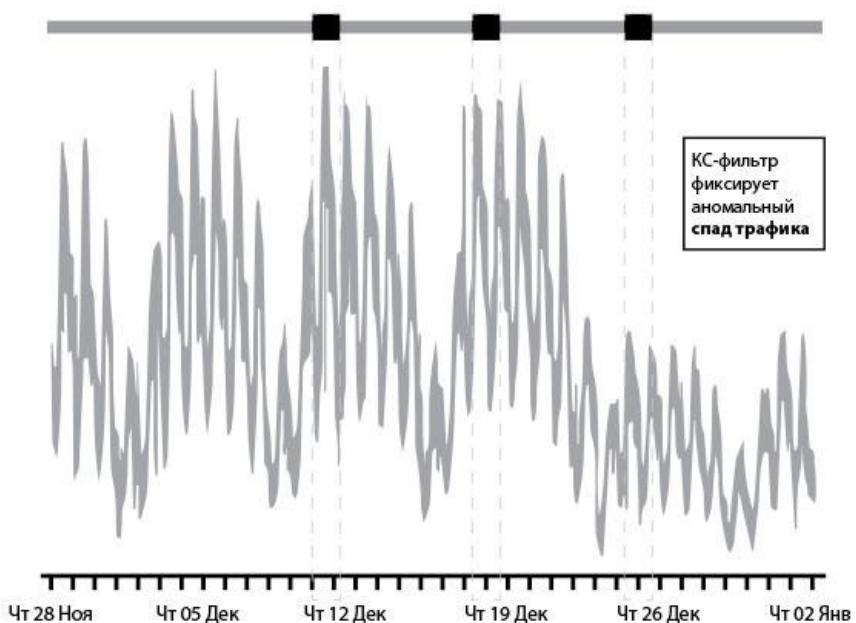


Рис. 36. Количество транзакций: использование теста Колмогорова — Смирнова для оповещения об аномалиях (источник: Туфик Бубе, "Simple math for anomaly detection")

В этой главе мы исследовали несколько разных статистических методик. Их можно использовать для анализа производственной телеметрии, чтобы заблаговременно находить и устранять возможные проблемы, пока они достаточно малы и еще не успели привести к катастрофическим последствиям. Эти методики позволяют находить слабые сигналы о неполадках. На их основе можно предпринять реальные действия, создать более безопасную систему, а также успешно добиваться поставленных целей.

Мы разобрали конкретные случаи деятельности реальных компаний, например то, как Netflix использовал эти подходы для проактивного удаления плохих вычислительных серверов и автоматического масштабирования своей вычислительной инфраструктуры. Мы также обсудили, как использовать скользящее среднее и тест Колмогорова — Смирнова. Его можно найти в популярных руководствах для построения графиков.

В следующей главе мы опишем, как интегрировать производственную телеметрию в повседневную деятельность команды разработки, чтобы сделать развертывание более безопасным и улучшить всю систему в целом.

## **Глава 16. Настройте обратную связь, чтобы разработчики и инженеры эксплуатации могли безопасно разворачивать код**

В 2006 г. Ник Галбрет был техническим директором в компании Right Media, отвечавшим и за разработку, и за эксплуатацию одной платформы онлайн-рекламы, отображавшей и обслуживавшей более 10 миллионов показов в день.

Галбрет так описывает конкурентную среду, где ему приходилось функционировать:

«В нашем бизнесе рекламные инструменты крайне изменчивы, поэтому нам нужно отвечать на запросы рынка за считанные минуты. Это означает, что отдел разработки должен вносить изменения в код очень быстро и переводить их в эксплуатацию как можно скорее, иначе мы проиграем более быстрым конкурентам. Мы обнаружили, что наличие отдельной группы для тестирования и даже для развертывания сильно тормозило процесс. Нужно было соединить все эти функции в одной группе, с общей ответственностью и общими целями. Хотите верьте, хотите нет, но самой большой трудностью оказался страх разработчиков перед развертыванием их собственного кода!»

В этом есть ирония: разработчики часто жалуются, что инженеры эксплуатации боятся развертывать код. Но в этом случае, получив возможность внедрять собственный код, разработчики точно так же боялись проводить развертывание.

Такой страх не есть нечто необычное. Однако Галбрет отметил: более быстрая и частая обратная связь для проводящих развертывание сотрудников (будь то разработчики или отдел ИТ-эксплуатации), а также сокращение размеров новой порции работы создали атмосферу безопасности и уверенности.

Наблюдая за множеством команд, прошедших через такую трансформацию, Галбрет так описывает последовательность изменений:

«Начинаем мы с того, что никто в разработке или эксплуатации не хочет нажимать на кнопку “развертывание кода”. Эту кнопку мы разработали специально для автоматизации процесса внедрения кода. Страх, что именно ты окажешься тем, из-за кого сломается вся производственная система, парализует всех сотрудников до единого. В конце концов находится смелый доброволец, решавшийся первым вывести свой код в производственную среду. Но из-за неверных предположений или невыявленных тонкостей первое развертывание неизбежно идет не так гладко, как хотелось бы. И, поскольку у нас нет достаточной телеметрии, о проблемах мы узнаем от заказчиков».

Чтобы исправить проблему, наша команда в спешном порядке исправляет код и направляет его на внедрение, но на этот раз сопровождая его сбором телеметрии в приложениях и в окружении. Так мы можем убедиться, что наши поправки восстановили работоспособность системы и в следующий раз мы сможем отследить подобную ошибку до того, как нам об этом сообщат клиенты.

Далее, все больше разработчиков начинают отправлять свой код в эксплуатацию. И, поскольку мы действуем в довольно сложной системе, мы наверняка что-то сломаем. Но теперь мы сможем быстро понять, какая именно функциональность вышла из строя, и быстро решить, нужно ли откатывать все назад или же можно решить все меньшей кровью. Это огромная победа для всей команды, все празднуют — мы на коне!

Однако команда хочет улучшить результаты развертывания, поэтому разработчики стараются проактивно получить больше отзывов от коллег о своем коде (это описано в ), все помогают друг другу писать лучшие автоматизированные тесты, чтобы находить ошибки до развертывания. И, поскольку все теперь в курсе, что чем меньше вносимые изменения, тем меньше проблем у нас будет, разработчики начинают все чаще посыпать все меньшие фрагменты кода в эксплуатацию. Таким образом, они убеждаются, что эти изменения проведены вполне успешно и можно переходить к следующей задаче.

Теперь мы развертываем код чаще, чем раньше, и надежность сервиса тоже становится лучше. Мы заново открыли, что секрет спокойного и непрерывного потока — в небольших и частых изменениях. Изучить и понять их может любой сотрудник.

Галбрет отмечает, что такое развитие событий идет на пользу всем, включая отдел разработки, эксплуатации и службу безопасности. «Поскольку я отвечаю и за безопасность тоже, мне становится спокойно от мысли, что мы можем быстро развертывать поправки в коде, потому что изменения внедряются на протяжении всего рабочего дня. Кроме того, меня всегда поражает, насколько все инженеры становятся заинтересованы в безопасности, когда ты находишь проблему в их коде и они могут быстро исправить ее сами».

История компании Right Media демонстрирует, что недостаточно просто автоматизировать процесс развертывания — мы также должны встроить мониторинг телеметрии в процесс внедрения кода, а

также создать культурные нормы, гласящие, что все одинаково ответственны за хорошее состояние потока создания ценности.

В этой главе мы создадим механизмы обратной связи, помогающие улучшить состояние потока создания ценности на каждой стадии жизненного цикла сервиса или услуги, от проектирования продукта и его разработки до развертывания, сопровождения и снятия с эксплуатации. Такие механизмы помогут нам убедиться, что наши сервисы всегда «готовы к эксплуатации», даже если они находятся на ранних стадиях разработки. Кроме того, такой подход поможет встроить новый опыт после каждого релиза и после каждой проблемы в нашу систему, и работа станет для всех и безопаснее, и продуктивнее.

На этом шаге мы убедимся в том, что мы ведем активный мониторинг всей телеметрии, когда кто-нибудь проводит развертывание, как было продемонстрировано в истории Right Media. Благодаря этому после отправки нового релиза в эксплуатацию любой сотрудник (из разработки, из отдела эксплуатации) сможет быстро определить, все ли компоненты взаимодействуют так, как нужно. В конце концов, мы никогда не должны рассматривать развертывание кода или изменение продукта, пока оно не будет осуществляться в соответствии с производственной средой.

Мы добьемся цели благодаря активному мониторингу показателей нужного элемента функциональности во время развертывания кода. Так мы убедимся, что мы в продукте ничего случайно не сломали или — что еще хуже — что мы ничего не сломали в другом сервисе. Если изменения портят программу или наносят ущерб ее функциональности, мы быстро восстанавливаем ее, привлекая нужных для этого специалистов.

Как было описано в части III, наша цель — отловить ошибки в процессе непрерывного развертывания до того, как они дадут о себе знать в эксплуатации. Однако все равно будут ошибки, и мы их упустим, поэтому мы полагаемся на телеметрию, чтобы быстро восстановить работоспособность сервисов. Мы можем либо отключить неисправные компоненты функциональности с помощью флагов (часто это самый простой и безопасный способ, поскольку он не требует развертывания), либо *закрепиться на передовой* (*fix forward*) (то есть переписать код, чтобы избавиться от проблемы; этот код затем отправляется в эксплуатацию), либо *откатывать изменения назад* (*roll back*) (то есть возвращаться к предыдущему релизу с помощью флагов-переключателей или удаления сломанных сервисов с помощью таких стратегий развертывания, как Blue-Green и канареевые релизы и так далее).

Хотя закрепление на передовой часто рискованно, оно может быть очень надежным, если у нас есть автоматизированные тесты и процессы быстрого развертывания, а также достаточно телеметрии для быстрого подтверждения, что в эксплуатации все осуществляется правильно.

На рис. 37 показано развертывание кода на PHP в компании Etsy. После окончания оно выдало предупреждение о превышении времени исполнения. Разработчик заметил проблему через несколько минут, переписал код и отправил его в производство. Решение проблемы заняло меньше десяти минут.

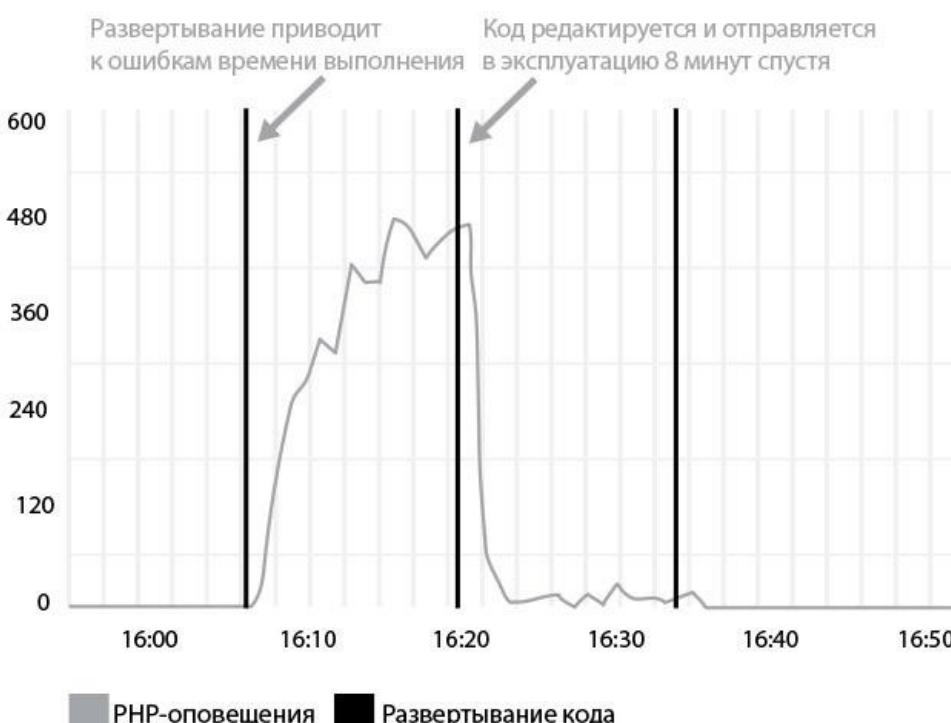


Рис. 37. Разворачивание кода в генерирует оповещения об ошибках времени выполнения. Они исправляются довольно быстро (источник: Майк Бриттен, "Tracking every release")

Поскольку производственное развертывание — одна из главных причин неполадок в эксплуатации, информация о каждом внедрении и внесении изменений отображается на графиках, чтобы все сотрудники в потоке создания ценности были в курсе происходящих процессов. Благодаря этому улучшаются координация и коммуникация, а также быстрее замечаются и исправляются ошибки.

Даже если развертывания кода и релизы проходят гладко, в любой сложной системе всегда будут неожиданные проблемы, например сбои и падения в самое неподходящее время (каждую ночь в 2:00). Если их оставить без исправления, они будут источником постоянных проблем для инженеров в нижней части потока. Особенно если неполадки не видны инженерам в верхней части, ответственным за создание задач с проблемами.

Даже если устранение неполадки поручают команде разработчиков, его приоритет может быть ниже, чем создание новой функциональности. Проблема может возникать снова и снова на протяжении недель, месяцев и даже лет, приводя к хаосу и дестабилизации эксплуатации. Это хороший пример того, как команды в верхней части потока создания ценности могут оптимизировать работу для себя и в то же время ухудшить производительность всего потока создания ценности.

Чтобы не допустить этого, сделаем так, чтобы все исполнители в потоке создания ценности делили ответственность по устранению сбоев в эксплуатации. Этого можно добиться, если разработчики, руководители разработки и архитекторы будут участвовать в дежурствах. Именно так в 2009 г. поступил Педро Канауати, директор по организации производства компании Facebook. Все сотрудники получат конкретную и практическую обратную связь по любым принятым решениям в отношении архитектуры или кода.

Благодаря такому подходу инженеры эксплуатации не остаются один на один с проблемами эксплуатации, возникшими из-за плохого кода. Вместо этого все, вне зависимости от положения в потоке создания ценности, помогают друг другу найти подходящий баланс между исправлением проблем на стадии эксплуатации и разработкой новой функциональности. Как заметил Патрик Лайтбоди, SVP по управлению продукцией компании New Relic, «мы обнаружили, что, если будить разработчиков в два часа ночи, дефекты исправляются как никогда быстро».

Побочный эффект такой практики — то, что менеджмент разработки начинает понимать: бизнес-цель достигнута не тогда, когда напротив нового элемента функциональности ставится пометка «сделано». Вместо этого элемент считается выполненным, когда работает в эксплуатации именно так, как задумывалось, без чрезмерного количества оповещений о проблемах или незапланированной работы для девелоперов и отдела эксплуатации.

Такая методика подходит и для ориентированных на рынок команд, отвечающих за разработку функциональности и за ввод ее в эксплуатацию, и для команд, ориентированных только на разработку приложения. Как в своей презентации 2014 г. отметил Аруп Чакрабарти, менеджер по эксплуатации в компании PagerDuty, «компании все реже и реже содержат специальные дежурные команды; вместо этого любой сотрудник, притрагивающийся к коду или окружению, должен быть доступен во время сбоя».

Независимо от того, как мы организуем команды, основополагающие принципы остаются все теми же: когда разработчики получают обратную связь, как работают их приложения, в том числе участвуя в устранении неполадок, они становятся ближе к заказчику. А это ведет к задействованности всех участников потока создания ценности.

Одна из самых эффективных методик по взаимодействию с пользователем и проектированию пользовательского интерфейса (UX) — это исследование в контексте (contextual inquiry). Суть этой методики в том, что команда разработчиков наблюдает, как заказчик использует приложение в естественной среде, зачастую прямо на своем рабочем месте. В результате часто открываются пугающие подробности, с каким трудом пользователи пронираются сквозь дебри интерфейса: десятки кликов для выполнения простейших ежедневных задач, копирование вручную или перемещение текста между несколькими окнами, записывание важной информации на обычных бумажках. Все это — примеры компенсирующего поведения и обходных решений там, где удобство и практичность должны быть в первую очередь.

Самая частая реакция разработчиков после такого исследования — смятение. «Ужасно видеть многочисленные способы расстроить наших клиентов». Такие наблюдения почти всегда приводят к значительным открытиям и жгучему желанию улучшить результат для заказчика.

Наша цель — использовать ту же самую методику для понимания, как наша деятельность влияет на

внутренних заказчиков. Разработчики должны отслеживать ситуацию со своим кодом на протяжении всего потока ценности, чтобы понимать, как коллеги справляются с результатами их работы при вводе кода в эксплуатацию.

Разработчики сами хотят следить за судьбой своего кода: видя трудности пользователей, они принимают более взвешенные и эффективные решения в повседневной работе.

Благодаря этому мы создаем обратную связь по нефункциональным аспектам кода — всем компонентам, не связанным с пользовательским интерфейсом, — и находим способы, как улучшить процесс развертывания, управляемость, удобство эксплуатации и так далее.

Исследование в контексте часто производит большое впечатление на заказчиков. Описывая свое первое наблюдение за работой пользователя, Жене Ким, соавтор этой книги и основатель компании Tripwire, проработавший в ней 13 лет на должности СТО, отмечает:

Один из худших моментов в моей профессиональной карьере произошел в 2006 г. Я целое утро наблюдал, как один из наших заказчиков работал с нашим продуктом. Он выполнял операцию, которую, как мы думали, клиенты будут выполнять еженедельно, и, к нашему ужасу, ему потребовалось 63 клика. Представитель заказчика все время извинялся: «Простите, наверняка есть способ сделать это лучше».

К несчастью, способа лучше не имелось. Другой клиент рассказал, что установка и настройка нашего продукта заняли у него 1300 шагов. Внезапно я понял, почему работа по сопровождению нашего приложения всегда доставалась новичкам в команде: никто не хотел брать это на себя. Такова одна из причин моего участия в создании методики наблюдения за работой пользователей в моей компании: я хотел искупить вину перед нашими заказчиками.

Наблюдение в контексте улучшает качество итогового продукта и развивает эмпатию по отношению к коллегам. В идеале эта методика помогает создавать четкие нефункциональные требования, позволяя проактивно встраивать их в каждый сервис или продукт. Это важная часть создания рабочей культуры DevOps.

Даже когда разработчики каждый день пишут и проверяют код в среде, похожей на среду заказчика, у эксплуатации все равно могут случаться неудачные релизы, потому что это все равно первый раз, когда мы на самом деле видим, как ведет себя код в реальных эксплуатационных условиях. Такой результат возможен потому, что новый опыт в цикле разработки ПО наступает слишком поздно.

Если оставить это без внимания, то готовый продукт становится трудно сопровождать. Как однажды сказал один анонимный инженер из отдела эксплуатации, «в нашей группе большинство системных администраторов не задерживалось дольше, чем на шесть месяцев. В эксплуатации все время что-то ломалось, рабочие часы превращались в безумие, развертывание приложений было невероятно болезненным. Самое худшее — объединение серверных кластеров приложения, на что у нас иногда уходило по шесть часов. В такие моменты мы думали, что разработчики нас ненавидят».

Это может быть результатом недостаточного количества ИТ-инженеров, поддерживающих все команды по разработке конкретного продукта и все программное обеспечение, уже находящееся в эксплуатации. Такое возможно и в командах, ориентированных непосредственно на рынок, и в командах, чья цель — разработка функциональности.

Одна из возможных мер — сделать как Google: там все группы разработчиков сами поддерживают и управляют своими сервисами, пока они не перейдут к централизованной группе инженеров эксплуатации. Если разработчики сами будут отвечать за развертывание и сопровождение своего кода, переход к централизованной команде эксплуатации будет гораздо более гладким.

Чтобы сомнительные самоуправляемые сервисы не попадали в эксплуатацию и не создавали ненужных рисков для организации, можно задать четкие требования для запуска готового приложения. Кроме того, чтобы помочь командам, разрабатывающим продукт, инженеры эксплуатации должны время от времени консультировать их, чтобы сервисы точно были готовы к внедрению.

Создавая ориентиры по запуску продукта, мы удостоверимся, что все команды извлекают пользу из коллективного опыта организации, особенно опыта отдела эксплуатации. Эти ориентиры и требования, скорее всего, будут включать в себя следующее.

- **Количество дефектов и их серьезность.** Работает ли приложение так, как было запланировано?
- **Тип и частота оповещений о проблемах.** Не выдает ли приложение чрезмерное количество

оповещений на стадии эксплуатации?

- **Уровень мониторинга.** Достаточен ли уровень мониторинга для восстановления работоспособности сервиса, если что-то идет не так?
- **Системная архитектура.** Достаточно ли автономны компоненты сервиса для того, чтобы поддерживать высокий темп изменений и развертываний?
- **Процесс развертывания.** Есть ли предсказуемый, четкий и достаточно автоматизированный процесс развертывания кода в эксплуатацию?
- **Гигиена эксплуатации.** Достаточно ли доказательств того, что процесс сопровождения хорошо организован, чтобы можно было быстро и без проблем передать его в другие руки?

На первый взгляд эти требования похожи на традиционные контрольные перечни, использованные ранее. Однако ключевое различие в том, что требуется наличие эффективного мониторинга, процесс развертывания должен быть четким и надежным, а архитектура — поддерживать быстрое и частое внедрение кода.

Если во время оценки найдены недостатки, инженер эксплуатации должен помочь команде разработчиков решить проблемы и при необходимости даже перепроектировать продукт, чтобы его было легко развернуть и сопровождать.

Возможно, на этом этапе стоит проверить, соответствует ли наш сервис нормативным требованиям или должен ли он будет соответствовать в будущем.

- Приносит ли продукт достаточно дохода (например, если доход составляет более 5 % общего дохода открытого акционерного общества компании в США, он считается «значительной частью» и подлежит регулированию в соответствии с разделом 404 Акта Сарбейнза — Оксли (SOX) 2002 г.)?
- Высок ли у приложения пользовательский трафик? Высокая ли у него цена простоя? Например, могут ли проблемы эксплуатации повлечь за собой репутационные издержки или риски доступности?
- Хранит ли сервис информацию о владельцах платежных карт, например номера таких карт, или какую-либо личную информацию, например номера социального страхования или записи о лечении пациентов? Есть ли проблемы с безопасностью, влекущие за собой нормативные, репутационные риски, риски по выполнению контрактных обязательств или по нарушению права на сохранение личной информации?
- Есть ли у сервиса какие-нибудь другие нормативные или контрактные обязательства, например в связи с правилами экспорта, стандартами безопасности данных индустрии платежных карт (PCI-DSS), законом об ответственности и переносе данных о страховании здоровья граждан (HIPAA)?



Рис. 38. Service Handback (Передача управления приложением), Google (источник: «SRE@Google: Thousands of DevOps Since 2004», видео с сайта YouTube, 45:57, выложено пользователем USENIX, 12 января 2012 г., )

Эта информация поможет проверить, что мы управляем не только техническими рисками, связанными с нашим продуктом, но также и потенциальными рисками, связанными с нормативным регулированием и обеспечением безопасности. Подобные соображения также будут полезными при проектировании среды контроля над производством продукции.

Благодаря встраиванию требований к эксплуатации в ранние стадии процесса разработки и переносу ответственности за первоначальное сопровождение приложений на разработчиков процесс внедрения новых продуктов становится более гладким, простым и предсказуемым. Однако, если сервис уже находится в эксплуатации, нужен другой механизм, чтобы отдел эксплуатации не оказался один на один с неподдерживаемым приложением. Это особенно важно для функционально ориентированных организаций.

В этом случае можно создать *механизм возвращения сервиса разработчикам*. Другими словами, когда продукт становится достаточно неустойчивым, у эксплуатации есть возможность вернуть ответственность за его поддержку обратно разработчикам.

Когда сервис переходит под крыло разработчиков, роль отдела эксплуатации меняется на консультационную: теперь они помогают команде снова сделать продукт готовым к эксплуатации.

Этот механизм служит подобием парового клапана, чтобы инженеры эксплуатации никогда не оказывались в такой ситуации, что им нужно поддерживать нестабильно работающее приложение, в то время как все увеличивающееся количество долгов погребает их с головой и локальные проблемы превращаются в одну глобальную. Эта методика также гарантирует, что у эксплуатации всегда есть время на улучшение и оптимизацию работы и на разные профилактические меры.

Механизм возврата — одна из практик-долгожительниц компании Google и, пожалуй, отличный способ показать взаимное уважение между инженерами разработки и эксплуатации. С его помощью разработчики могут быстро создавать новые сервисы, а отдел эксплуатации может подключаться, когда продукты становятся стратегически важными для компании, и в редких случаях возвращать их обратно, когда поддерживать становится слишком проблематично. Следующий пример из практики подразделения по обеспечению стабильности сайтов компании Google показывает, как эволюционировали проверки на готовность к смене сопроводителя и на готовность к запуску и какую пользу это принесло.

Один из многих удивительных фактов о компании Google заключается в том, что ее инженеры эксплуатации ориентированы на функциональность. Этот отдел именуют обеспечением надежности сайтов (Site Reliability Engineering, SRE), название было придумано Беном Трейнором Слоссом в 2004 г. В этом году он начал работать с семью инженерами, а в 2014 г. в отделе работало уже более 1200 сотрудников. По его словам, «если Google когда-нибудь пойдет ко дну, это будет моя

вина». Трейнор Слосс сопротивлялся всем попыткам как-то определить, что же такое SRE, но однажды описал это подразделение так: SRE — это то, «что происходит, когда разработчик занимается тем, что обычно называли эксплуатацией».

Каждый инженер SRE находится в подчинении организации Трейнора Слосса, чтобы качество персонала всегда было на должном уровне. Они есть в каждой команде компании, обеспечивающей их финансирование. Однако SRE-инженеров так мало, что их приписывают только к тем командам, чья работа важнее всего для организации или должна соответствовать специальным нормативным требованиям. Кроме того, у этих сервисов должна быть низкая операционная нагрузка. Продукты, не соответствующие этим критериям, остаются под управлением разработчиков.

Даже когда новые продукты становятся достаточно важными для компании, чтобы можно было передать их SRE, разработчики должны сопровождать свои продукты как минимум в течение шести месяцев, прежде чем к команде можно будет прикрепить SRE-инженеров.

Чтобы самостоятельные команды все же могли воспользоваться опытом организации SRE, Google создал два списка проверок для двух важнейших стадий релиза нового сервиса. Это проверка на готовность к передаче, к смене сопроводителя (Hand-Off Readiness Review, HRR), и проверка на готовность к запуску продукта (Launch Readiness Review, LRR).

Проверка на готовность к запуску должна проводиться перед введением в эксплуатацию любого нового сервиса Google, тогда как проверка на готовность к смене сопроводителя проводится в том случае, когда сервис передается под контроль эксплуатации, обычно через несколько месяцев после запуска. Чек-листы (контрольные списки) процессов LRR и HRR похожи, но проверка на смену управления более строгая, ее стандарты выше, в то время как проверку перед внедрением устраивают сами команды.

Всем командам, проходящим через любую проверку, помогает инженер SRE. В его задачи входит помочь им понять требования и выполнить их. Со временем чек-листы изменялись и дополнялись, и теперь любая команда может извлечь пользу из коллективного опыта всех предыдущих запусков, как успешных, так и не очень. Во время своей презентации «SRE@Google: тысячи DevOps с 2004 г.» Том Лимончелли отметил: «Каждый раз, выпуская новый продукт, мы учимся чему-то новому. Всегда будут те, у кого меньше опыта в организации релизов и запусков. Чек-листы LRR и HRR — хороший способ создать память компании».

Благодаря требованию управлять своими собственными сервисами на стадии эксплуатации разработчики могут понять, каково это — работать в эксплуатации. Списки LRR и HRR не только делают передачу управления более простой и предсказуемой, но также помогают развивать эмпатию между сотрудниками в начале и конце потока создания ценности.

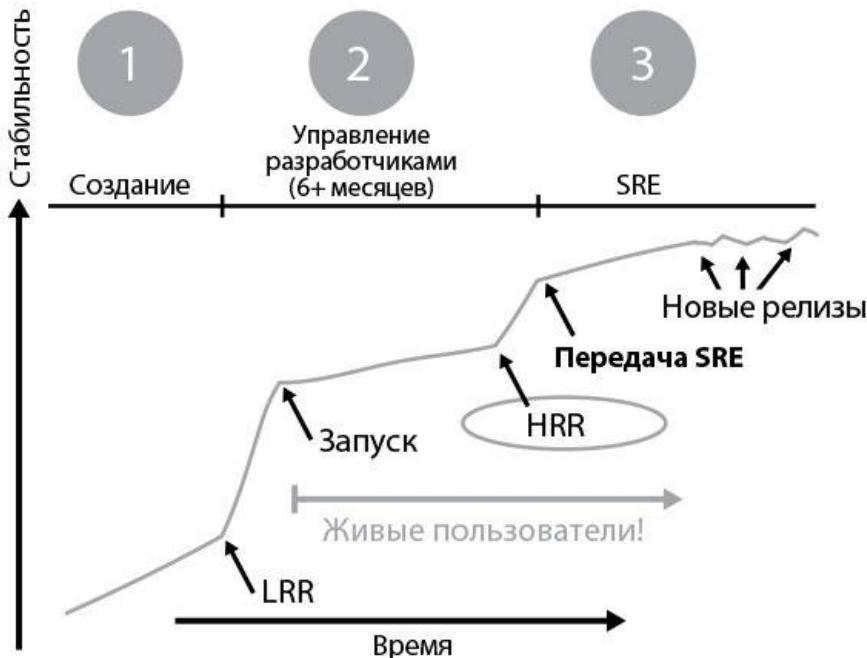


Рис. 39. Проверка на готовность к запуску и проверка на готовность к смене сопроводителя в компании Google (источник: "SRE@Google: Thousands of DevOps Since 2004", видео с сайта YouTube, 45:57, выложено пользователем USENIX, 12 января 2012 г., )

Лимончелли отмечает: «Согласно идеальному сценарию, команды разработчиков используют чек-лист LRR в качестве руководства, работая над соблюдением его требований параллельно с

разработкой своего продукта. Если же им нужна помощь, они консультируются с инженерами SRE».

Кроме того, Лимончелли замечает: «Команды, быстрее всего добивающиеся одобрения проверки HRR, работают с инженерами SRE дольше всего, с начальных этапов проектирования и до запуска сервиса. Самое хорошее то, что получить помощь сотрудника SRE всегда очень просто. Каждый инженер ценит возможность дать совет команде на ранних этапах и, скорее всего, добровольно выделит только для этого несколько часов или дней».

Обычай инженеров SRE помогать командам на первых стадиях проекта — важная культурная норма, постоянно укрепляемая в Google. Лимончелли объясняет: «Помощь разработчикам — долговременная инвестиция, приносящая плоды много месяцев спустя, когда настает время запуска. Эта форма “социально ответственной позиции” и “работы на благо общества” здесь очень ценится. Ее регулярно рассматривают, когда оценивают SRE-инженеров на предмет возможного повышения».

В этой главе мы обсудили механизмы обратной связи, позволяющие улучшить продукты на каждом этапе ежедневной работы, будь то изменения в процессе развертывания, исправление кода после сбоев и срочных вызовов инженеров, обязанности разработчиков отслеживать состояние своего кода в конце потока создания ценности, создание нефункциональных требований, помогающих разработчикам писать более подходящий для эксплуатации код, или возвращение проблематичных сервисов разработчикам на доработку.

Создавая такие цепи обратной связи, мы делаем развертывание кода более безопасным, увеличиваем степень его готовности к эксплуатации и помогаем улучшить отношения между разработчиками и эксплуатацией, укрепляя понимание общих целей, общей ответственности и эмпатии.

В следующей главе мы исследуем, как телеметрия может помочь основанной на выдвижении гипотез разработке и A/B-тестированию и как проводить эксперименты, помогающие быстрее добиться целей компании и завоевывать рынок.

## **Глава 17. Встройте основанную на гипотезах разработку и А/В-тестирование в свою повседневную работу**

Слишком часто в проектах разработки ПО новый функционал создается в течение нескольких месяцев или лет без всякого подтверждения достижения требуемых бизнес-целей с каждым новым релизом. Например, служит ли конкретный функционал желаемым целям, или даже используется ли он вообще.

Еще хуже, если внесение изменений в компонент, который не работает, может быть отодвинуто на задний план разработкой нового функционала в приложении. Неэффективный компонент так никогда и не будет действовать на полную мощность. В общем случае, как отмечает Джез Хамбл, «самый неэффективный способ протестировать бизнес-модель или идею продукта — создать этот продукт и посмотреть, есть ли на него спрос».

Прежде чем мы встраиваем новую функциональную возможность, мы должны строго спросить себя: а надо ли нам создавать ее и почему? Далее нужно провести наиболее быстрые и дешевые эксперименты, чтобы с помощью исследования пользовательских мнений подтвердить, будет ли реальный компонент приложения соответствовать запланированному результату. Для этого можно использовать такие методики, как разработка на основе гипотез, воронки привлечения клиентов и А/В-тестирование. Эти методики мы более подробно изучим в данной главе. Компания Intuit Inc. — отличный пример того, как организация использует подобные методики для создания популярных продуктов, распространения опыта внутри компании и завоевания своего рынка.

Intuit специализируется на создании продуктов для финансового менеджмента и делового администрирования. Цель — упростить жизнь малого бизнеса, покупателей и профессиональных бухгалтеров. В 2012 г. доход компании составлял 4,5 миллиарда долларов, в ней работали около 8500 сотрудников. Ее основными продуктами были QuickBooks, TurboTax, Mint и до недавнего времени Quicken.

Скотт Кук, основатель Intuit, всегда был сторонником культуры инноваций, вдохновляя команды на экспериментальный подход к разработке программ и убеждая руководство поддерживать их в этом. По его словам, «вместо того чтобы следовать мнению своего босса, стоит сконцентрироваться на поведении реальных участников реального эксперимента и уже затем на основе результатов принимать решения». Это отличный пример научного подхода в разработке ПО.

Кук объясняет, что необходима «такая система, где каждый сотрудник может проводить быстрые эксперименты... Дэн Морер управляет нашим подразделением по работе с потребителями, а оно, в свою очередь, занимается сайтом TurboTax. К тому моменту, как он получил эту должность, мы устраивали примерно семь экспериментов в месяц».

Далее он продолжает: «Создав процветающую культуру инноваций в 2010 г., теперь они проводят по 165 экспериментов в три месяца периода подачи налоговых деклараций. Бизнес-результаты? Показатель конверсии сайта — около 50 %... Членам команды это нравится, потому что теперь их идеи работают на реальном рынке».

Помимо роста показателя конверсии одна из самых интересных деталей этой истории — что в отделе TurboTax проводили эксперименты в периоды максимальной нагрузки на сайт. Десятилетиями в розничной торговле риск влияющих на выручку сбоев во время праздничных сезонов был настолько велик, что с середины октября по середину января мы просто запрещали вносить любые изменения.

Однако, добившись быстрых и безопасных развертываний и релизов, команда TurboTax снизила риски пользовательских экспериментов и внесения изменений настолько, что их можно было проводить и в самые прибыльные периоды наибольшей нагрузки.

Это подчеркивает мысль, что самое ценное для экспериментов время — как раз месяцы и дни интенсивной нагрузки. Если бы команда TurboTax ждала до 16 апреля, когда заканчиваются сроки подачи налоговой документации в США, компания потеряла бы много потенциальных и реальных клиентов: они ушли бы к конкурентам.

Чем быстрее мы экспериментируем, воспроизводим результаты и встраиваем их в наш продукт или сервис, тем быстрее учимся и тем сильнее превзойдем конкурентов. А скорость интегрирования результатов зависит от организации развертывания и выпуска релизов.

Пример компании Intuit показал, что команда TurboTax смогла повернуть ситуацию себе на пользу и в результате захватила рынок.

Как показывает история команды TurboTax, определение воронки привлечения клиентов и А/В-тестирование — очень мощные инструменты исследования поведения пользователей. Методики А/В-тестирования появились в *прямом маркетинге* — одном из двух основных направлений

маркетинговых стратегий. Второе направление — *массовый маркетинг*, или *бренд-маркетинг*, — часто заключается в том, чтобы разместить перед потенциальными покупателями как можно больше рекламы.

В прошлые эпохи, до электронной почты и социальных сетей, прямой маркетинг сводился к рассылке тысяч открыток и буклетов обычной почтой и к просьбам возможным клиентам принять предложение и перезвонить по указанному телефону, выслать ответную открытку или оставить заказ каким-либо другим способом.

В рамках таких кампаний исследования проводились для того, чтобы определить способ с наибольшим показателем конверсии. Они экспериментировали с изменением подачи предложения, меняя слова, стили, дизайн, оформление, упаковку и так далее, все для того, чтобы понять, как наилучшим образом вызвать желаемые действия покупателя (например, чтобы он позвонил по телефону или заказал товар).

Зачастую каждый эксперимент требовал дизайна и печати нового тиража, рассылки тысяч предложений, ждать ответов приходилось неделями. Каждая попытка обычно обходилась в десятки тысяч долларов и занимала несколько недель или месяцев. Однако, несмотря на все расходы, такое повторяющееся тестирование вполне окупалось, если оно значительно увеличивало показатель конверсии (например, если процент заказавших продукт покупателей увеличивался с 3 до 12 %).

Хорошо задокументированные примеры А/В-тестирования включают в себя сбор средств, интернет-маркетинг и методологию бережливого стартапа. Интересно, что эта методика также использовалась британским правительством для определения того, с помощью каких писем эффективнее всего было собирать просроченные налоги с задерживающих платежи граждан.

Самая распространенная методика А/В-тестирования в современной практике взаимодействия с пользователем — сайт. На нем посетителям показывается одна из двух страниц, контрольная («А») и альтернативная («В»), выбранная случайным образом. На основе статистического анализа делаются выводы о значимости различий в поведении двух групп пользователей. После этого мы можем установить причинно-следственную связь между внесенным изменением (например, в функциональных возможностях, элементах дизайна, цветах фона) и последствиями (например, коэффициентом конверсии, средним размером заказа).

К примеру, можно провести такой эксперимент: проверить, увеличивает ли доход изменение цвета кнопки «купить». Или замерить, уменьшает ли выручку рост времени ответа сайта (с помощью искусственного замедления его работы). Такой тип А/В-тестирования позволит нам оценить улучшение работоспособности сайта в денежном выражении.

А/В-тесты также известны под такими названиями, как «онлайн-эксперименты в контролируемых условиях» или «сплит-тесты». Эксперименты можно проводить и с несколькими переменными. Благодаря этому мы сможем наблюдать их взаимодействие. Такая методика называется многомерным тестированием.

Результаты А/В-тестирования часто просто поразительны. Ронни Кохави, выдающийся инженер и директор отдела анализа и проведения экспериментов компании Microsoft, заметил: после «оценки тщательно продуманных и поставленных экспериментов, проведенных с целью улучшить какой-либо ключевой показатель, выяснилось, что только одна треть действительно помогла улучшить этот показатель!». Другими словами, воздействие двух третей новых компонентов функциональности было либо незначительным, либо вовсе отрицательным. Кохави отмечает, что все эти новшества казались хорошими и обоснованными идеями, что еще раз подчеркивает преимущество пользовательского тестирования перед интуицией и экспертными оценками.

Выходы из данных Кохави ошеломляют. Если мы не исследуем поведение пользователей, вероятность того, что разрабатываемая нами функциональность бесполезна или наносит вред компании, равна двум третям. И это не считая того, что сам код усложняется, увеличиваются затраты на его поддержку, а вносить изменения становится все труднее. Кроме того, усилия, затраченные на создание таких элементов функциональности, могли бы быть потрачены на что-то действительно полезное (то есть это альтернативные издержки). Джез Хамбл в связи с этим пошутил: «Если довести эту мысль до крайности, то компании и заказчикам было бы выгоднее отправить всю команду в отпуск, чтобы она не занималась разработкой бесполезных компонентов».

Лекарство от этого — интеграция А/В-тестирования в проектирование, разработку, тестирование и развертывание элементов функциональности. Осмыщенное исследование поведения пользователя и постановка экспериментов помогут нам достичь намеченных целей и занять прочные позиции на рынке ПО.

Быстрое и многократное А/В-тестирование возможно в том случае, если мы можем быстро и без усилий развертывать наш код, используя переключатели функциональности для отключения ненужных компонентов и отправляя в эксплуатацию несколько версий кода одновременно для

разных сегментов заказчиков. Для этого нужна полезная телеметрия на всех уровнях стека приложения.

С помощью переключателей мы можем контролировать процент пользователей, участвующих в эксперименте. Например, пусть одна половина будет контрольной группой, а другая будет видеть следующее предложение: «Заменить в вашей корзине товары, в данный момент отсутствующие, на похожие». Мы будем сравнивать поведение экспериментальной группы (видящей предложение) и контрольной (не видящей), измеряя, например, количество покупок за одну сессию.

Компания Etsy выложила в открытый доступ свою экспериментальную программу Feature API (ранее известную как Etsy A/B API), поддерживающую не только A/B-тестирование, но и онлайн-регулирование количества пользователей из контрольной группы. Среди других инструментов для A/B-тестирования можно назвать Optimizely, Google Analytics и др.

В интервью 2014 г. с Кендрисом Вонгом из компании Apptimize Лейси Роадс так описал свой путь: «Экспериментирование в Etsy происходит из желания принимать обоснованные решения и гарантировать то, что, когда мы запускаем новую функциональность для миллионов наших пользователей, она действительно работает. Очень часто у нас появлялись такие программные компоненты, разработка и поддержание которых отнимали много времени и сил, а никаких доказательств их популярности у пользователей не имелось. A/B-тестирование позволяет нам... еще на стадии разработки понять, что над этой функциональностью действительно стоит работать».

Когда выстроена инфраструктура, поддерживающая A/B-релизы и тестирование, нужно убедиться, что представители заказчика думают о каждом компоненте функциональности как о гипотезе и используют релизы для экспериментов с реальными пользователями, чтобы подтвердить или опровергнуть гипотезы. Планирование эксперимента должно проходить в контексте воронки привлечения клиентов. Барри О'Райли, соавтор книги *Lean Enterprise: How High Performance Organizations Innovate at Scale*, описывает, как можно формулировать гипотезы в разработке возможностей приложения:

**«Мы верим**, что увеличение размеров фотографий отелей на сайте бронирования **приведет к увеличению интереса клиентов и повышению показателя конверсии. Мы будем уверены в необходимости продолжения, если количество клиентов, просматривающих изображения отелей и затем в течение 48 часов бронирующих номер, увеличится на 5 %».**

Исследовательский подход к разработке требует не только разбивать задачу на небольшие подзадачи (требования или пожелания пользователей), но также убеждаться, что каждая подзадача приводит к ожидаемому исходу. Если этого не происходит, мы исправляем план работ и ищем другие пути.

Чем быстрее мы сможем встраивать обратную связь в продукт или сервис, поставляемый нашим клиентам, тем быстрее будем учиться и тем значительнее окажутся результаты нашей деятельности. То, насколько сильно короткие циклы влияют на результат работы, можно проследить на примере компании Yahoo! Answers, когда она перешла от одного релиза каждые шесть недель к нескольким релизам за одну неделю.

В 2009 г. Джим Стоунхэм занимал должность директора подразделения Yahoo! Communities. В него входили Flickr и Answers. До этого он отвечал за работу Yahoo! Answers. Конкурентами были такие сервисы вопросов и ответов, как Quora, Aardvark и Stack Exchange.

В то время у сервиса Answers было примерно 140 миллионов посетителей в месяц, свыше 20 миллионов активных пользователей, отвечающих на вопросы на более чем 20 языках. Однако рост доходов и числа пользователей прекратился, а показатели активности пользователей сокращались.

Стоунхэм отмечает: «Yahoo! Answers был и остается одной из самых больших социальных игр во всем интернете. Десятки миллионов участников активно пытаются набрать уровни, давая качественные ответы быстрее, чем другие пользователи сайта. У нас было много возможностей подправить игровую механику, петли виральности и другие способы взаимодействия в сообществе. Когда имеешь дело с человеческим поведением, нужно уметь устраивать быстрые тесты, чтобы увидеть, что пришлось людям по душе».

Стоунхэм продолжает: «Эти эксперименты очень хорошо делали компании Twitter, Facebook и Zynga. Они устраивали эксперименты как минимум дважды в неделю, даже разбирали и анализировали изменения, сделанные до развертывания, чтобы убедиться, что они все еще на правильном пути. Вот он — я, управляющий крупнейшим сайтом вопросов и ответов на всем рынке, желающий внедрить быстрое и многократное тестирование функциональности. Но мы не можем выпускать релизы чаще раза в четыре недели. А другие на этом же рынке получают обратную связь в десять раз быстрее, чем мы».

Стоунхэм отмечает, что сколько бы заказчики и разработчики ни говорили о важности телеметрии,

без частых экспериментов (каждый день или каждую неделю) они всего лишь фокусируются на компонентах функциональности, а не на результатах клиента.

Когда команда Yahoo! Answers смогла перейти на еженедельное развертывание, а затем и на несколько развертываний в неделю, ее способность экспериментировать с функциональными возможностями сильно выросла. Впечатляющие достижения и новый опыт за следующие 12 месяцев регулярных экспериментов увеличили количество посещений на 72 %, активность пользователей возросла втрое, а команда удвоила доход. Чтобы укрепить успех, она сосредоточилась на оптимизации следующих показателей.

- Время до первого ответа. Как быстро появился ответ на вопрос пользователя?
- Время до лучшего ответа. Как быстро сообщество присудило награду за лучший ответ?
- Количество плюсов на один ответ. Сколько плюсов было поставлено за ответ?
- Вопросы/неделя/пользователь. Сколько ответов давали пользователи?
- Уровень повторного поиска. Как часто посетители были вынуждены повторять поиск, чтобы найти нужный ответ? (Чем ниже, тем лучше.)

Стоунхэм подводит итог: «Это были как раз знания, необходимые, чтобы завладеть рынком, — и они изменили не только скорость наших компонентов функциональности. Из команды наемных работников мы превратились в команду заказчиков. Когда двигаешься с такой скоростью и каждый день смотришь на числа и результаты, уровень вовлеченности в работу меняется радикально».

Для успеха нужны не только быстрое внедрение и быстрые релизы, но и умение побороть конкурентов в проведении экспериментов. Такие методики, как разработка на основе гипотез, определение и измерение воронки привлечения клиентов и А/В-тестирование, позволяют безопасно и без усилий экспериментировать с поведением пользователей, пробуждая творческий и инновационный подход к работе и создавая новый опыт на уровне всей компании. И хотя преуспеть — важно, новые знания, полученные благодаря экспериментам, дают работникам контроль над целями организации и удовлетворенностью клиентов. В следующей главе мы рассмотрим и создадим процессы проверки, анализа и координации для улучшения качества текущей работы.

## Глава 18. Создайте процессы проверки и координации для улучшения качества текущей работы

В предыдущих главах мы создали систему телеметрии, необходимую для обнаружения и решения проблем в эксплуатации и на всех стадиях процесса непрерывного развертывания, а также быстрые циклы обратной связи от клиентов для получения нового опыта — опыта, пробуждающего вовлеченность в процесс и ответственность за удовлетворенность клиентов и работу всех компонентов сервиса. Все это помогает добиваться успеха.

Цель этой главы — помочь разработчикам и отделу эксплуатации сократить риск производственных изменений до того, как они сделаны. Обычно, анализируя изменения перед новым развертыванием, мы полагаемся на отзывы, проверки и согласование прямо перед самым развертыванием. Часто согласование дается чужими командами, слишком далекими от нашей работы. Принять обоснованное решение о рискованности правок затруднительно, а время на получение всех разрешений удлиняет сроки внесения изменений.

Процесс проверки работы коллегами (peer review) сервиса GitHub — яркий пример того, как критическое рассмотрение может улучшить качество, сделать развертывание более безопасным и стать частью повседневной работы. Специалисты сервиса GitHub стали первопроходцами практики pull request (запроса на внесение изменений), одной из самых популярных форм проверки кода коллегами как в разработке, так и в эксплуатации.

Скотт Чейкон, директор по информационным технологиям и один из основателей компании GitHub, писал на своем сайте, что запросы на внесение изменений — механизм, позволяющий инженерам сообщать об изменениях, отправляемых в репозиторий GitHub. Когда запрос отправлен, заинтересованные лица могут оставить отзыв о предлагаемых правках, обсудить потенциальные модификации и даже, если потребуется, предложить расширение кода на основе предлагаемых изменений. Программисты, отправляющие запрос, часто запрашивают «+1», «+2» и так далее в зависимости от того, сколько отзывов нужно.

На GitHub запросы на внесение изменений — также механизм для развертывания кода в производственную среду через набор практик, называемых пользователями «потоком GitHub». Так программисты запрашивают рецензии, собирают обратную связь, на ее основе вносят изменения в код и сообщают о развертывании кода в производственной среде (то есть в ветку master).

The screenshot shows a GitHub pull request interface. On the left, there's a vertical list of line numbers (35, 36, 37, 38) corresponding to the code. Lines 27 through 31 are highlighted in grey. Line 27 contains the code '+ opts[:options] [:stripnl] || = false'. Lines 28, 29, 30, and 31 show the addition of code for handling timeouts and Pygments highlighting. A comment from user 'brianmario' is visible, asking about defaults if no encoding or lexer is passed. Another comment from 'Josh' says he'll push the changes down to colorize them. At the bottom, there's a button labeled 'Add a line note'.

```
35  27 + opts[:options] [:stripnl] || = false
36  28
37  29
38  30
39  31     timeout opts.delete(:timeout) || DEFAULT_TIMEOUT do
          begin
            Pygments.highlight(text, opts)
```

brianmario repo collab

So what are the defaults here if no encoding or lexer is passed?

Also there's at least one other place where the API is expected to take an :encoding key (not nested under an :options key/hash) -  
<https://github.com/github/github/blob/master/app/models/gist.rb#L114>

Only reason I did it that way was to sorta abstract the fact that we're using pygments for colorizing currently (not that we have plans to change that anytime soon...)

Josh repo collab

Alright, I'll push that down to colorize.

Add a line note

Рис. 40. Комментарии и предложения к запросу внесение изменений, GitHub (источник: Скотт Чикон, “GitHub Flow”, сайт , 31 августа 2011 г., )

Поток GitHub состоит из пяти этапов.

1. Чтобы начать работать над чем-то новым, инженер создает соответствующую именованную ветку от «master» (например, «new-oauth2-scopes»).
2. Инженер работает только с этой веткой, регулярно отправляя свой код на сервер.
3. Когда ему нужна обратная связь или помочь или когда он считает, что ветка готова к слиянию с master, он отправляет запрос на внесение изменений.
4. Когда инженер получает рецензии и необходимые согласования новой функциональности, он может добавить свои изменения в ветку «master».
5. Когда правки внесены и отправлены в главную ветку, инженер развертывает их в производственную среду.

Эти практики, встраивающие рецензирование и координацию в процесс ежедневной работы, позволили компании GitHub быстро и надежно поставлять на рынок продукты высокого качества и высокой надежности. Например, в 2012 г. сотрудники провели 12 062 развертывания — потрясающее количество. В частности, 23 августа на общекорпоративной конференции придумали и обсудили много новых замечательных идей, и это был самый загруженный день этого года: сотрудники и пользователи провели 563 сборки и 175 успешных развертываний. Все благодаря механизму запросов на внесение изменений.

В этой главе мы будем постепенно разбирать практики, используемые в GitHub, чтобы избежать зависимости от периодических проверок и согласований и перейти к комплексному и непрерывному рецензированию кода в процессе ежедневной работы. Наша цель — сделать так, чтобы разработка, ИТ-эксплуатация и служба информационной безопасности перешли к модели постоянного сотрудничества, чтобы вносимые в наши системы изменения работали надежно, безопасно и так, как было запланировано.

Провал компании Knight Capital — одна из самых показательных ошибок внедрения ПО в недалеком прошлом. Пятнадцатиминутная ошибка развертывания — причем инженеры не могли отключить рабочие сервисы — привела к потере 440 миллионов долларов. Финансовые убытки поставили под удар функционирование организации, в результате чего владельцы были вынуждены быстро продать компанию, чтобы она могла продолжать деятельность без риска для финансовой системы.

Джон Оллспоу замечает, что, когда происходят такие резонансные провалы, как ошибка развертывания Knight Capital, обычно есть две противоречащие фактам интерпретации, почему же эти сбои вообще могли случиться.

Первая интерпретация гласит, что авария произошла из-за сбоя в системе контроля изменений. Звучит правдоподобно, поскольку мы можем представить ситуацию, когда лучшие методики контроля могли бы отследить такой риск раньше и не допустили бы отправки изменений в производственную среду. А если бы они и не смогли предотвратить развертывание плохого кода, то мы быстрее обнаружили бы ошибку и сразу же предприняли шаги по восстановлению системы.

Вторая интерпретация заключается в том, что провал случился по вине сбоя тестирования. Это тоже кажется правдоподобным. Чем лучше методы тестирования, тем больше шансы отследить неполадки в самом начале и отменить рискованное развертывание. Или как минимум мы могли бы быстрее заметить ошибку и вернуть систему в рабочее состояние.

Удивительно, но факт: в компаниях с низким уровнем доверия и культурой, основанной на контроле и управлении, частый результат жесткого контроля над изменениями и тестированием — высокая вероятность того, что такие проблемы повторятся снова, с еще более печальным исходом.

Джин Ким (соавтор этой книги) называет осознание того, что контроль над изменениями и тестирование могут иметь совершенно противоположный результат, «одним из важнейших моментов моей профессиональной карьеры. Это озарение было результатом одного разговора в 2013 г. с Джоном Оллспоу и Джезом Хамблом о провале Knight Capital. Я начал сомневаться в некоторых своих основополагающих убеждениях, сформированных за последние десять лет, особенно если учсть мое аудиторское образование».

Ким продолжает: «Как бы печально это ни было, этот момент был для меня ключевым. Они не только убедили меня в том, что они правы, но мы также протестирували эти предположения в обзоре 2014 State of DevOps Reports, что привело ко многим потрясающим открытиям. Построение культуры с высоким уровнем доверия — наверное, самый большой вызов для менеджмента в этом десятилетии».

Традиционный контроль над изменениями может привести к непредусмотренным результатам, например к долгим срокам разработки продукта, сокращению силы и скорости обратной связи с этапа развертывания кода. Чтобы понять, как это происходит, рассмотрим часто применяемые средства управления во время провала контроля над изменениями:

- добавление новых вопросов в форму запроса на внесение изменений;
- требование большего числа разрешений. Например, еще один уровень согласования в управлеченческой иерархии (допустим, вместо разрешения директора по эксплуатации теперь нужно разрешение от директора по информационным технологиям) или других заинтересованных лиц (например, системных администраторов, комиссий контроля по архитектуре и так далее);
- требование большего количества времени для разрешения на внесение изменений, чтобы правки оценивались и анализировались как должно.

Такие методы контроля часто добавляют помехи в процесс развертывания, увеличивая количество лишних шагов, нужных согласований и время развертывания. Все это, как мы знаем, сокращает вероятность благоприятного итога и для разработчиков, и для отдела эксплуатации. Излишний контроль также замедляет обратную связь о качестве нашей работы.

Один из важнейших постулатов системы производства Toyota — «самые близкие к проблеме исполнители обычно знают о ней больше всего». Это особенно заметно, если выполняемая работа или рабочая система становятся более сложной и динамичной, что характерно как раз для потоков создания ценности DevOps. В таких случаях потребность в разрешении руководителей, находящихся все дальше и дальше от текущей задачи, может уменьшить вероятность благоприятного исхода. Как уже не раз было доказано, чем больше расстояние между сотрудником, выполняющим работу (то есть тем, кто вносит какие-либо изменения), и человеком, принимающим решение о ее необходимости (то есть тем, кто выдает разрешение на внесение изменений), тем хуже результат.

В обзоре 2014 State of DevOps Reports компании Puppet Labs одним из ключевых открытий было то, что высокоеэффективные организации больше полагались на рецензии коллег, а не на внешнее одобрение изменений. На рис. 41 показано: чем больше компаний зависят от получения разрешений, тем хуже их результаты и в отношении стабильности (среднее время восстановления сервиса и доля неудачных изменений), и в отношении производительности (время на развертывание, частота развертываний).

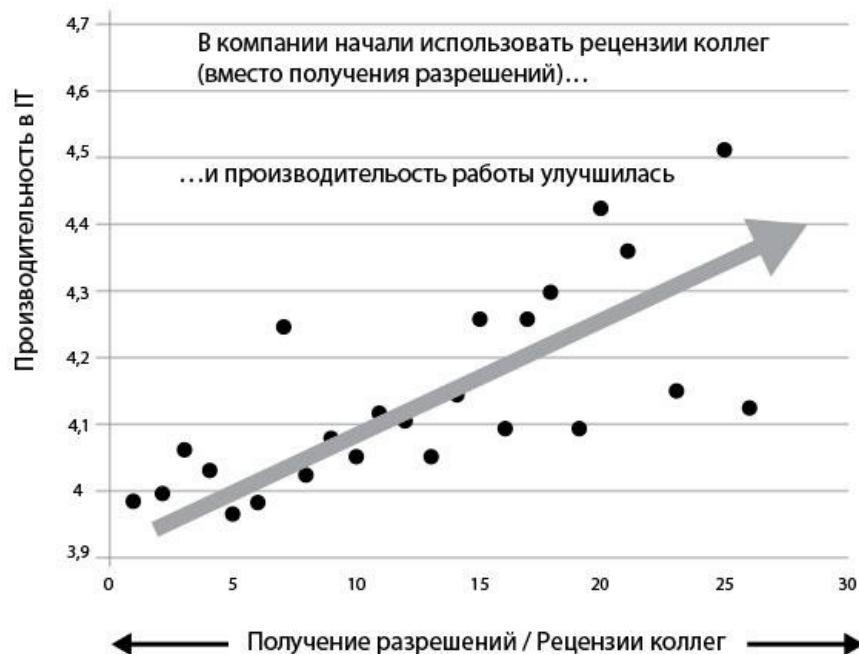


Рис. 41. Производительность организаций с рецензированием кода выше, чем у организаций, где требуется разрешение на внесение изменений (источник: отчет DevOps Survey of Practice, 2014 г., компания Puppet Labs)

Во многих организациях консультативные советы по изменениям играют важную роль в координации и управлении процессом внедрения, но их работа не должна сводиться к оценке каждой правки кода. Стандарт ITIL тоже не требует такого подхода.

Чтобы понять, почему так происходит, представьте, в каком затруднительном положении оказываются члены консультативных советов. Им нужно рецензировать сложные изменения, состоящие из тысяч строк кода, что написаны сотнями программистов.

Одна крайность в том, что, просто читая пространное описание правок или проверяя все пункты чек-листа, мы не сможем предсказать, будут ли изменения успешными. Другая — в том, что тщательный осмотр под микроскопом тысяч строк кода вряд ли раскроет что-то новое. Одна из причин этого — сама природа изменений в сложных системах. Даже инженеры, каждый день работающие с базой исходного кода, часто бывают удивлены побочными эффектами того, что казалось обычной правкой с низким уровнем риска.

Все эти причины свидетельствуют, что нужно создать эффективные практики контроля, более близкие к рецензированию кода коллегами и сокращающие зависимость от внешних органов, одобряющих или запрещающих внесение изменений. Кроме того, нам нужно эффективно координировать и планировать изменения. Этой темой мы и займемся в двух следующих параграфах.

Всякий раз, когда над взаимозависимыми системами работают несколько групп, необходимо как-то согласовывать вносимые изменения, чтобы они не влияли друг на друга (например, группируя их по какому-либо принципу или выстраивая в очередь). В общем случае чем более независимы компоненты архитектуры, тем меньше нам нужно общаться и координироваться с другими командами. Когда архитектура — действительно сервис-ориентированная, группы разработчиков могут вносить изменения с большей долей автономии. Локальные правки вряд ли станут причиной глобальных сбоев.

Однако даже в слабо связанной архитектуре, когда много команд проводят сотни независимых развертываний в день, есть риск того, что изменения будут мешать друг другу (например, при одновременном А/В-тестировании). Для уменьшения этих рисков можно использовать чаты, чтобы оповещать о планируемых правках и проактивно обнаруживать возможные накладки.

Для более сложных систем и систем с сильно связанный архитектурой возникает необходимость планирования изменений: представители команд собираются вместе, но не для того, чтобы одобрить правки или выдать на них разрешение, а чтобы составить расписание и порядок внесения изменений для снижения числа возможных неприятностей.

Однако в некоторых областях, таких как изменения в глобальной инфраструктуре (например, исправления в коммутаторе ядра сети), риск всегда будет выше. Для них нужны будут специальные профилактические меры: избыточность, перехват управления при отказе, комплексные испытания и (в идеале) симуляция.

Вместо того чтобы требовать разрешения на развертывание у внешнего арбитра, можно побуждать инженеров рецензировать работу своих коллег. В разработке эта практика носит название *анализ кода*, но ее можно приспособить и для любых других изменений в приложениях или окружении, включая серверы, сеть и базы данных. Суть метода в том, чтобы кто-нибудь из инженеров, близких к конкретной задаче, тщательно рассмотрел и проанализировал предлагаемые правки. Такое рецензирование улучшает качество вносимых исправлений, а также стимулирует взаимное обучение и улучшение навыков.

Логичный момент, когда нужно проводить анализ, — перед отправкой кода в систему управления кодом, где изменения потенциально могут привести к глобальному сбою. Как минимум рецензирование должен проводить ваш непосредственный коллега, но для областей с более высоким уровнем риска, например для баз данных или критичных для бизнеса компонентов со слабым покрытием автоматизированными тестами, может потребоваться более глубокий анализ эксперта (например, инженера по информационной безопасности или инженера по базам данных) или же несколько разных рецензий от разных специалистов («+2» вместо «+1»).

Принцип небольших кусков кода применяется и в анализе кода. Чем больше размер разбираемого фрагмента кода, тем больше времени нужно, чтобы его понять, и тем больше ответственность на проверяющем сотруднике. Как отмечает Рэнди Шуп, «есть нелинейная зависимость между величиной вносимых изменений и потенциальным риском внедрения этих изменений: когда вы переходите от десяти строк кода к сотне, риск вырастает больше чем в десять раз, и так далее». Поэтому так важно, чтобы разработчики были заняты небольшими, постепенными приращениями кода, а не одной многолетней веткой компонента функциональности.

Кроме того, наша способность осмысленно критиковать правки кода уменьшается с ростом объема этих правок. Как написал в своем твиттере Гирей Озил, «попросите программиста проверить десять

строчек кода, и он найдет десять проблем. Попросите его проверить сотню строчек, и он скажет, что все хорошо».

Принципы рецензирования кода включают в себя следующее:

- у каждого сотрудника должен быть кто-нибудь, кто проверяет его исправления (и в коде, и в окружении) перед отправкой в основной код проекта;
- все сотрудники должны следить за потоком правок своих коллег, чтобы можно было заметить и проанализировать потенциальные конфликты изменений;
- определите, какие изменения считаются очень рискованными: они могут потребовать анализа эксперта в этой области (например, исправления в базах данных, такие важные с точки зрения безопасности модули, как подтверждение прав доступа, и так далее);
- если кто-то собирается внести слишком объемные для понимания правки — другими словами, если даже после нескольких прочтений вы не можете понять, к каким последствиям они приведут, или вам нужно просить о разъяснениях, — их нужно разделить на более мелкие части, понятные с первого взгляда.

Чтобы убедиться в том, что мы не просто механически штампует бессмысленные рецензии, можно проанализировать статистику по рецензиям, чтобы найти соотношение между принятыми и отвергнутыми правками. Возможно, полезно провести выборочное исследование конкретных рецензий.

Анализ кода может принимать несколько разных форм.

- **Парное программирование.** Программисты работают парами (см. раздел ).
- **«Чересплечное» программирование.** Один программист наблюдает за работой другого, когда тот проходит по написанному им коду.
- **Передача по электронной почте.** Система управления исходным кодом автоматически рассыпает код на рецензию после того, как программист ставит пометку о внесении изменений
- **Анализ кода с помощью инструментов.** Авторы и рецензенты используют специально разработанные для обзора кода инструменты (например, Gerrit, запросы на внесение изменений GitHub и так далее) или средства, предоставляемые репозиториями исходного кода (например, GitHub, Mercurial, Subversion, а также такие платформы, как Gerrit, Atlassian Stash и Atlassian Crucible).

Тщательный анализ изменений эффективен для обнаружения пропущенных ошибок. Рецензирование кода поможет увеличить частоту внесения правок и развертываний. Оно также поддерживает модель разработки и внедрения на основе единой ветви кода (*trunk-based development*) и непрерывной поставки. Рассмотрим это подробнее на следующем примере из практики.

Компания Google — прекрасный пример организации, использующей разработку на основе единой ветви кода и непрерывную масштабируемую поставку. Как отмечалось ранее, Эран Мессери рассказывал, что в 2013 г. организация процессов в Google позволяла более чем 13 000 разработчиков работать в своих ветвях над одним и тем же исходным кодом, совершая более 5500 коммитов в неделю. Число развертываний за неделю доходило до нескольких сотен. Согласно данным 2010 г., каждую минуту в основную ветку кода отправлялось более двадцати правок, из-за таких темпов каждый месяц обновлялось 50 % кода.

Для такой организации от команд разработчиков требуются значительная дисциплина и обязательное рецензирование кода, покрывающее следующие области:

- легкая читаемость кода (нужны руководства по стилю оформления кода);
- назначение ответственных за поддеревья кода для поддержания единообразия и контроля над отсутствием ошибок;

- прозрачность кода и сотрудничество в работе над одним и тем же кодом между разными командами.

На рис. 42 показано, как время на рецензирование кода зависит от размера внесенных изменений. На оси X отмечается размер правок, на оси Y — время, затраченное на проверку кода. В общем случае чем больше объем изменений, тем дольше проходит анализ. Точки в верхнем левом углу обозначают более сложные и потенциально рискованные изменения, требующие более глубокого обдумывания и обсуждения.

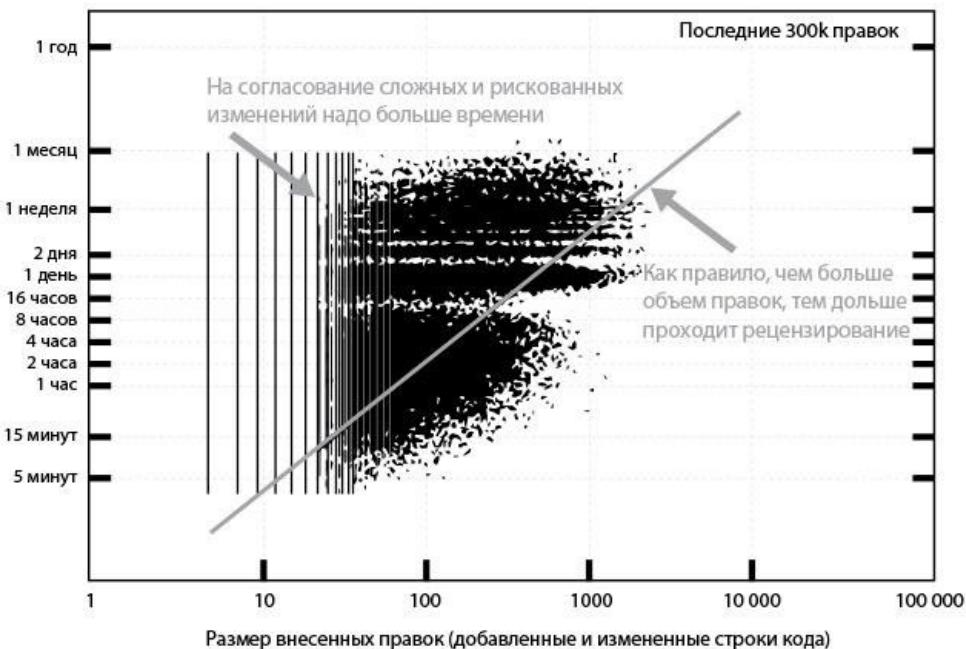


Рис. 42. Длительность рецензирования в зависимости от размера кода в компании Google (источник: Ашиш Кумар, "Development at the Speed and Scale of Google," презентация на QCon, Сан-Франциско, CA, 2010.)

Чтобы решить техническую проблему компании Google, Рэнди Шупп, в то время занимавший должность технического директора организации, начал свой личный проект. Он отмечал: «Я работал над этим проектом неделями и наконец решился попросить специалиста в этой области проверить мой код. Это было примерно три тысячи строк. Через них рецензент продирался несколько дней. После чего он попросил: "Пожалуйста, больше так не делай". Я был очень благодарен ему, что он нашел время помочь мне. Именно тогда я и понял, как встроить рецензирование кода в ежедневную работу».

Теперь, когда мы ввели рецензирование кода, уменьшающее риски, сокращающее время на введение изменений и способствующее непрерывной масштабируемой поставке, как мы выяснили на примере компании Google, перейдем к тому, как тестирование может обернуться против нас. Когда происходят ошибки тестирования, типичная реакция — провести еще больше тестов. Однако если мы просто проводим тесты в конце нашего проекта, то можем ухудшить результаты.

Это особенно верно, если мы проводим тестирование вручную, поскольку оно более медленно и трудоемко. «Дополнительное тестирование» обычно требует больше времени, чем предполагалось, что сокращает частоту развертываний и повышает объем единовременно вносимых правок кода. А мы уже знаем и из теории, и из практики, что большие объемы нового развертываемого кода снижают шансы на благоприятный исход и увеличивают значение MTTR и число сбоев — совсем не то, что нам нужно.

Вместо того чтобы тестировать объемные куски кода за счет замораживания всех других правок, мы хотим встроить тестирование в процесс повседневной работы как часть непрерывного потока на стадии внедрения, а также увеличить частоту развертываний. Благодаря этому система будет изначально качественной, что позволит тестировать, развертывать и выпускать код еще более мелкими порциями, чем раньше.

Парное программирование — ситуация, когда два программиста вместе работают на одном рабочем месте. Этот метод был популяризован в ранних 2000-х гг. в таких направлениях, как экстремальное программирование (Extreme Programming) и гибкая методология разработки (Agile). Как и

рецензирование кода, этот способ появился в разработке, но он также применим и к деятельности любого инженера в потоке создания ценности. Далее термины *парная работа* и *парное программирование* будут использоваться как синонимы, чтобы подчеркнуть, что этот метод подходит не только разработчикам.

В одном из наиболее частых подходов один инженер *ведущий* — он пишет код. Другой — *штурман, наблюдатель* или *наводчик* — анализирует и оценивает работу непосредственно во время выполнения. В процессе наблюдения он может обдумать стратегическое направление, придумать новые идеи улучшений и предугадать возможные проблемы. Это позволяет ведущему сосредоточить все внимание на тактических задачах, штурман страхует его от ошибок и указывает дальнейшее направление движения. Если у членов пары разные специальности, то побочный эффект — ненамеренное обучение друг друга навыкам как через ситуативную тренировку, так и с помощью обмена методиками и неочевидными способами решения проблем.

Еще один подход к парному программированию воплощает принципы основанной на тестировании разработки (*test-driven development, TDD*): один инженер пишет автоматизированные тесты, а второй — собственно приложение. Джек Этвуд, один из основателей сайта Stack Exchange, пишет: «Я все время думаю о том, что парное программирование — всего лишь рецензирование кода на стероидах... Преимущество парного программирования — в его захватывающей непосредственности: невозможно игнорировать рецензента, если он сидит рядом с тобой».

Джек Этвуд продолжает: «Большинство людей пассивно уклоняются от необходимости оценивать чужой код, если у них имеется такая возможность. В парном программировании это невозможно. Каждая половина пары должна понимать код, здесь и сейчас, прямо во время его написания. Работа в паре может идти через силу, но она также поднимает коммуникацию на высочайший уровень. Достичь этого иначе просто невозможно».

В 2001 г. Лори Уильямс провела исследование, где показала, что «программисты в паре работают на 15 % медленнее, а объем кода без ошибок увеличивается с 70 до 85 %. Поскольку тестирование и отладка часто во много раз дороже, чем создание первоначального кода, эти результаты впечатляют. Пары обычно рассматривают больше альтернатив, чем программисты-одиночки, и в результате их решения проще и поддерживать их легче. Они также раньше замечают ошибки проектирования». Лори Уильямс также отмечает, что 96 % ее респондентов сообщили, что работа в паре понравилась им больше, чем работа в одиночку.

У парного программирования есть дополнительное преимущество: оно способствует распространению знаний в компании и увеличивает обмен информацией в команде. Когда более опытные инженеры наблюдают за тем, как их менее опытные коллеги пишут код, обучение проходит гораздо эффективнее.

Элизабет Хендриксон, технический директор организации Pivotal Software, Inc. и автор книги *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*, много раз говорила, что каждая команда должна нести ответственность за качество своего труда, вместо того чтобы возлагать ответственность на отделы. Она утверждает, что такой подход не только повышает качество результата, но также увеличивает количество изменений.

В своей презентации на конференции DevOps Enterprise Summit в 2015 г. она рассказала, что в 2011 г. в Pivotal было два метода оценки кода: парное программирование (каждая строчка кода проверялась двумя людьми) и рецензирование кода с помощью программного обеспечения Gerrit (каждая фиксация кода должна была получить «+1» от двух человек, прежде чем отправиться в основной код проекта).

Элизабет заметила, что с Gerrit есть одна проблема: иногда программисты ждали рецензии целую неделю. Что хуже, у опытных программистов «появлялось раздражение и опускались руки оттого, что они не могли внести в код простейшие изменения, потому что мы непреднамеренно создали невыносимые заторы в работе».

Она пожаловалась: «Старшие инженеры единственные способны ставить «+1» правкам кода, но у них много других обязанностей, им часто все равно, над какими проблемами бьются младшие разработчики и какая у них производительность. Получилась ужасная ситуация: пока ты ждешь рецензии на изменения, другие разработчики отправляют код в систему. Целую неделю приходилось загружать внесенные другими изменения на свой ноутбук, еще раз прогонять все тесты, чтобы убедиться, что все удачно, и иногда отправлять код на рецензию заново!»

Чтобы решить эту проблему и устраниТЬ все эти задержки, в компании решили полностью избавиться от рецензирования кода в Gerrit. Вместо него для внесения правок в систему нужно было использовать парное программирование. Благодаря этому программисты сократили время на оценку кода с недель до нескольких часов.

Элизабет Хендриксон тем не менее отмечает, что рецензирование кода отлично работает во многих

организациях. Однако оно требует наличия культуры: анализ кода должен цениться так же высоко, как и его создание. Когда такой культуры еще нет, парное программирование может служить важным промежуточным шагом.

Поскольку рецензирование кода — важное средство контроля, мы должны уметь определять, эффективно оно или нет. Один метод заключается в том, чтобы рассмотреть сбои и оценить, можно ли что-то изменить в процессе рецензирования, чтобы этих сбоев можно было избежать.

Другой способ был предложен Райаном Томайко, директором по информационным технологиям и сооснователем GitHub, а также одним из авторов идеи запросов на внесение изменений. Когда его попросили описать разницу между хорошим запросом на внесение изменений и плохим, он сказал, что результаты не имеют к этому никакого отношения. У плохого запроса недостаточно контекста для читателя и нет описания, что эта правка должна делать, например, если сопроводительный текст выглядит как «Исправление проблемы #3616 и #3841».

Этот реальный внутренний запрос организации GitHub Райан Томайко раскритиковал: «Скорее всего, его написал один из новых сотрудников нашей компании. Во-первых, в нем не было указано ни одного инженера. Как минимум программист должен указать своего наставника или специалиста в той области, к которой относится правка, чтобы рецензию провел компетентный человек. Но вдобавок нет никакого пояснения, что это за изменение, почему оно важно или как размышлял его автор».

С другой стороны, на вопрос о хорошем запросе, отражающем эффективный процесс рецензирования, Томайко быстро перечислил важнейшие элементы: должно быть достаточно деталей о том, зачем нужно это изменение, как оно было проделано, а также его возможные риски и меры противодействия этим рискам».

Райан Томайко также обращает внимание на хороший анализ вносимого изменения в контексте запроса: часто в нем акцентируются дополнительные риски, идеи по более подходящим способам внедрения изменений и по смягчению возможных рисков и так далее. И если во время развертывания происходит что-то плохое или неожиданное, информация добавляется в запрос вместе со ссылкой на проблему. Обсуждение проходит без поиска виноватых; вместо него проходит открытый и непредвзятый разговор о том, как предотвратить такие проблемы в будущем.

В качестве примера Томайко приводит другой внутренний запрос GitHub касательно миграции баз данных. Он был длиной в несколько страниц, содержал обширный анализ возможных рисков и завершался следующей фразой автора запроса: «Я сейчас отправляю эту правку в исходный код. Сборки этой ветки сейчас падают, потому что на CI-серверах (серверах непрерывной интеграции) не хватает одного столбца» (ссылка на лог падения MySQL).

Автор правки потом извинился за сбой и рассказал, какие условия и ошибочные предположения привели к неполадке, а также предложил список мер для предотвращения этой проблемы в будущем. Все это сопровождалось страницами разбора и анализа. Читая этот запрос, Томайко улыбнулся: «Вот это отличный запрос!»

Как описано выше, мы можем оценить эффективность процессов рецензирования, проанализировав выборочные запросы как из всей совокупности запросов на внесение изменений, так и только из имевших отношение к сбоям.

Пока что мы обсуждали рецензирование кода и парное программирование, позволяющие увеличить качество результатов и избавиться от необходимости получать разрешения от внешних структур. Однако у многих компаний до сих пор есть процессы одобрения изменений. На них требуются месяцы. Эти процессы могут значительно увеличивать время создания продукта, что не только не дает быстро выполнять заказы клиентов, но также потенциально препятствует достижению целей организации. Если такое все же происходит, нужно изменить процессы внутри компаний, чтобы цели достигались быстрее и безопаснее.

Как отмечает Адриан Коккрофт, «вывешивать на широкое обозрение стоит отличный показатель — сколько встреч и тикетов нужно, чтобы сделать один релиз. Цель — неуклонно сокращать усилия инженеров на создание продукта и поставку его заказчикам».

Точно так же Тапабрата Пал, сотрудник компании Capital One, описал одну программу под названием Got Goo?. Специальная команда устраняет всевозможные препятствия, включая инструменты, процессы и одобрение, мешающие завершению процесса. Джейсон Кокс, старший директор по проектированию систем компании Disney, в презентации на конференции DevOps Enterprise Summit 2015 г. описал программу Join the Rebellion. Ее цель — избавить от тяжелого труда и всевозможных препятствий.

В 2012 г. в компании Target комбинация процесса принятия новых технологий (TEAP) и ведущей комиссии контроля по архитектуре (LARB) привела к тому, что на одобрение использования новой

технологии требовалось очень много времени. Чтобы предложить новую технологию, например иную базу данных или систему мониторинга, нужно было заполнить форму TEAP. Эти предложения затем оценивались, и стоящие заносились в список вопросов ежемесячных встреч LARB.

Хизер Микман и Росс Клэнтон, директор по разработке и директор по эксплуатации в Target, Inc. соответственно, помогали внедрять методы DevOps в своей компании. Микман нашла необходимую для одного проекта технологию (в данном случае Tomcat и Cassandra). Решение LARB гласило, что в данный момент команда эксплуатации не может ее поддерживать. Однако Микман была настолько убеждена в необходимости этих систем, что она предложила, чтобы за поддержку, интеграцию, доступность и безопасность этой технологии отвечала ее команда разработчиков, а не команда эксплуатации.

«Пока это предложение рассматривалось, я хотела понять, почему процесс TEAP-LARB занимает столько времени, и я использовала методику “пяти почему”... Она в конце концов привела меня к вопросу, зачем вообще нужен TEAP-LARB. Удивительно, но никто не мог на него ответить, кроме разве что расплывчатых соображений о том, что нам нужен какой-то контроль. Многие знали, что несколько лет назад произошла какая-то катастрофа и она ни за что не должна повториться снова. Но никто не мог точно вспомнить, что же это была за катастрофа».

Микман отмечает, что проходить через этот процесс ее группе было необязательно, если они соглашались быть ответственными за поддержание новой технологии. Она также добавляет: «Я дала всем знать, что все новые технологии, поддержанные моей командой, согласовывать с TEAP-LARB не нужно».

В результате систему управления базами данных Cassandra в Target успешно внедрили. Более того, процесс TEAP-LARB впоследствии был отменен. В благодарность за устранение барьеров для введения новых технологий команда Хизер Микман вручила ей награду за профессиональные достижения.

В этой главе мы обсудили, как вводить в работу новые методики, повышающие качество вносимых изменений, сокращающие риск неудачных развертываний и уменьшающие зависимость от процессов согласования и одобрения изменений. Примеры из практики компаний GitHub и Target показывают: эти методики не только улучшают наши результаты, но также значительно сокращают время на создание продуктов и увеличивают производительность разработчиков. Для всего этого требуется культура высокого доверия.

Вот история об одном младшем инженере, рассказанная Джоном Оллспоу. Этот инженер спросил, можно ли отправить в развертывание небольшую правку в HTML-коде, и Оллспоу ответил: «Я не знаю». Затем он спросил: «Кто-нибудь просмотрел твой код? Ты знаешь, кого лучше всего спрашивать об изменениях такого типа? Сделал ли ты все возможное, чтобы досконально убедиться, что эти изменения будут вести себя в эксплуатации именно так, как и планировалось? Если да, то не спрашивай меня — просто внеси правки, и все».

Таким ответом Оллспоу напомнил инженеру, что за качество изменений отвечает только он сам: если она сделал все, что мог, чтобы убедиться в том, что все работает правильно, тогда ему не нужно было просить одобрения, он мог просто внести свои правки.

Создание условий, когда авторы идей сами отвечают за их качество, — важная часть производительной культуры высокого доверия. Ее мы и хотим построить. Кроме того, такие условия позволяют создать более безопасную систему, где все помогают друг другу достичь целей, преодолевая любые препятствия.

В части IV мы показали, что петли обратной связи позволяют нам достигать общих целей, замечать проблемы на стадии их возникновения и с помощью быстрого обнаружения и восстановления гарантировать, что элементы функциональности не только работают так, как планировалось, но и способствуют выполнению целей организации и взаимному обучению внутри компании. Мы также изучили то, как общие цели становятся мостами через пропасть между разработкой и отделом эксплуатации, что оздоравляет весь поток создания ценности.

Теперь мы готовы перейти к части V: третий путь, методики обучения. В ней мы рассмотрим, как раньше других, быстро и дешево создавать возможности для обучения, чтобы можно было развернуть на полную мощность культуру инноваций и экспериментирования, позволяющую всем осмысленно действовать, способствуя преуспеванию организации.

## **Введение**

В части III, мы обсудили то, как использовать методики создания быстрого течения в потоке ценности. В части IV «Второй путь: методики обратной связи» нашей целью было создать как можно больше обратной связи — раньше, быстрее, дешевле, — покрывающей как можно больше частей нашей системы.

В части V «Третий путь: методики обучения» мы опишем методики, создающие подходящие возможности для обучения, настолько быстро, часто, дешево и рано, насколько это возможно. Сюда включаются извлечение нового опыта из неудач и провалов, неизбежных во взаимодействии со сложными системами, а также планирование и организация наших систем таким образом, чтобы мы могли постоянно экспериментировать и учиться новому, делая эти системы все более безопасными. В результате мы получим более развитые способности к адаптации и более глубокие знания, как на самом деле работают системы, что поможет нам быстрее двигаться к целям.

В следующих главах мы введем ритуалы, усиливающие безопасность и поощряющие непрерывное обучение и введение улучшений с помощью таких шагов, как:

- установление беспристрастной культуры, чтобы сотрудники чувствовали себя в безопасности;
- намеренное создание сбоев, чтобы улучшить способности к восстановлению;
- трансформация локальных открытых в глобальные улучшения;
- выделение времени на создание улучшений и новых знаний на уровне всей компании.

Мы также создадим специальные механизмы, чтобы новые знания, полученные в одной части организации, могли быстро распространяться по всей компании, тем самым превращая небольшие улучшения в масштабное продвижение вперед. Благодаря этому мы не только учимся быстрее конкурентов, отвоевывая у них рынок, но и создаем более безопасную и устойчивую культуру. В ней приятно работать, она максимально раскрывает человеческий потенциал.

## Глава 19. Внедрите обучение в повседневную работу

Когда мы работаем в сложной системе, предсказать все последствия наших действий невозможно. Часто это приводит к неожиданным и иногда катастрофическим последствиям, даже если мы пользуемся мерами предосторожности, например чек-листами или документацией, где фиксируем понимание системы на данный момент.

Для безопасной работы над сложными системами, организации должны совершенствовать процессы самодиагностики и внутренних улучшений, а также иметь развитые навыки обнаружения и устранения проблем. Это создает динамическую систему обучения, позволяющую понимать причины ошибок и переводить понимание в действия, предотвращающие повторение таких ошибок в будущем.

Такие организации доктор Стивен Спир называет эластичными. Они способны исцелять сами себя. «Для таких компаний реагирование на кризисы не есть нечто редкое и специфическое. Этим они занимаются все время. Таков источник их устойчивости».

Яркий пример отказоустойчивости, возникающей из следования этим принципам и методикам, продемонстрировал Netflix. 21 апреля 2011 г. вся зона доступности AWS US-EAST компании Amazon вышла из строя, захватив с собой всех зависящих от нее клиентов организаций, включая Reddit и Quora. Netflix, однако, оказался неожиданным исключением:казалось, что масштабный сбой AWS его не затронул.

Вслед за этим событием последовало множество домыслов о том, как Netflix смог удержать свои сервисы в рабочем состоянии. Популярная теория гласит, что, поскольку компания — один из крупнейших клиентов Amazon Web Services, у нее было привилегированное положение, что и позволило ей выстоять. Однако пост в блоге Netflix Engineering разъяснил, что причиной такой адаптивности компании оказались некоторые решения в планировании архитектуры, принятые еще в 2009 г.

В 2008 г. сервис поставки видео в режиме онлайн в Netflix работал на неделимом J2EE-приложении, расположенном в одном из его данных центров. Однако начиная с 2009 г. компания начала перестраивать архитектуру системы, адаптируя ее целиком под *облачные технологии* (*cloud native*): она была спроектирована так, чтобы работать в общедоступном облаке Amazon и быть достаточно гибкой, чтобы не падать при масштабных сбоях.

Одной из конкретных целей при планировании системы было условие, чтобы сервисы Netflix продолжали работать, даже если выйдет из строя вся зона доступности AWS, что и произошло с зоной US-EAST. Для этого архитектура системы должна была быть слабо связанной, а у каждого компонента должно было быть четкое время ожидания, чтобы из-за сбоя одного элемента не рухнула вся система. Вместо этого каждый элемент функциональности был спроектирован так, чтобы плавно деградировать производительность системы. Например, во время резкого увеличения трафика, создавшего повышенную нагрузку на CPU, персонализированная подборка рекомендуемых фильмов заменялась на статичное содержание — кэшированные или среднестатистические результаты, требующие гораздо меньших вычислений.

Кроме того, в посте блога рассказывалось, что, помимо внедрения новых архитектурных шаблонов, также построили и запустили неожиданный и дерзкий сервис Chaos Monkey, симулирующий сбои AWS, постоянно и в случайном порядке выводивший из строя серверы. Создатели хотели, чтобы все «команды инженеров привыкли к определенному количеству неполадок в облаке» и чтобы сервисы могли «автоматически восстанавливаться без вмешательства вручную».

Другими словами, с помощью Chaos Monkey и регулярных намеренных сбоев команда Netflix обрела уверенность, что цели адаптировать систему достигнуты.

Как можно было ожидать, во время первого запуска Chaos Monkey в эксплуатационном окружении сервисы выходили из строя так, как никто не мог предсказать и вообразить. Постоянно находя и устранивая эти проблемы во время обычных рабочих часов, инженеры Netflix быстро создали более устойчивый сервис и в то же время получили новый опыт (и это в рабочее время!), позволивший развить свои системы далеко за пределы того, что могли их конкуренты.

Chaos Monkey — далеко не единственный пример того, как обучение можно интегрировать в повседневную деятельность. Эта история также показывает, как ориентированные на обучение компании думают о неудачах, провалах и ошибках: здесь есть возможность научиться чему-то новому, а не найти, за что следует наказывать. В этой главе мы изучим, как создать ориентированную на обучение систему и развить культуру беспристрастности, а также как регулярно репетировать неполадки и намеренно создавать сбои, чтобы ускорить обучение.

Одна из предпосылок культуры обучения в том, что, когда сбои все-таки случаются (а они, без сомнения, неизбежны), реакция на них беспристрастна. Сидни Деккер, участвовавший в

определении ключевых элементов культуры безопасности и придумавший термин *беспрестрастная культура*, пишет: «Когда реакция на инциденты и ошибки воспринимается как небеспрестрастная, это может помешать расследованию причин. Вместо внимания и осознанности у исполнителей, занятых важной в плане безопасности работой, выращивается страх; вместо аккуратности и старательности в компаниях процветает бюрократизм, поощряются скрытность и забота только о себе».

Идея наказания скрыто или явно присутствует во многих методах, использованных менеджерами в прошлом столетии. По их представлениям, чтобы добиться целей компании, лидеры должны командовать, контролировать, устанавливать процедуры для устранения ошибок и принуждать к следованию этим процедурам.

Деккер называет желание избавиться от ошибок, избавившись от людей, совершивших эти ошибки, *теорией плохого яблока*. Он утверждает, что это неверный подход, потому что «человеческие ошибки — не причина наших проблем, а следствие проектирования инструментов, которые мы дали людям».

Если сбои возникают не из-за «плохих яблок», а из-за неизбежных ошибок проектирования сложных систем, то вместо поиска виноватых наша цель — увеличение возможностей для обучения и постоянное напоминание: мы ценим действия, помогающие выявлять проблемы в повседневной работе. Именно это повышает качество и безопасность систем, а также улучшает отношения между сотрудниками, задействованными в системе.

Превращая информацию в знание и встраивая результаты обучения в наши системы, мы строим культуру беспрестрастности, уравновешивая потребность в безопасности и ответственность. Как утверждает Джон Оллспоу, главный технический директор компании Etsy, «наша цель в Etsy — смотреть на ошибки, промахи, неудачи, провалы и тому подобное с точки зрения обучения».

Когда инженеры совершают ошибки и, рассказывая о них, чувствуют себя в безопасности, они не только хотят нести за них ответственность, но и горят желанием помочь всем остальным избежать этих ошибок в будущем. Именно это помогает распространять новые знания внутри компании. С другой стороны, если мы накажем этого инженера, никому не захочется сообщать важные детали, необходимые для понимания механизма и причин неполадки, а это гарантированно приведет к повторению этой ошибки.

Две эффективные методики, позволяющие создать беспрестрастную и ориентированную на обучение культуру, — это разбор ошибок без поиска виноватых и контролируемое создание сбоев в эксплуатации, чтобы можно было отрепетировать неизбежные в сложных системах проблемы. Сначала поговорим о разборе ошибок, после чего исследуем, почему ошибки могут оказаться благом для компаний.

Чтобы развить в организации культуру беспрестрастности, после аварий и значительных инцидентов (например, после неудачного развертывания или проблемы в эксплуатации, повлиявшей на клиентов), когда последствия сбоя уже устраниены, нужно провести «послеаварийную ретроспективу» (blameless post-mortem). Разбор ошибок без поиска виноватых (автор термина — Джон Оллспоу) помогает изучить «ошибку так, чтобы сфокусироваться на ситуационных аспектах механизма сбоя и на процессе принятия решений у человека, стоявшего ближе всех к сбою».

Для этого мы проводим совещание как можно быстрее после инцидента и до того, как связи между причинами и следствиями исчезнут из памяти или изменятся обстоятельства (конечно, мы ждем до того момента, когда проблема будет устранена, чтобы не отвлекать тех, кто над ней еще работает).

Во время совещания по разбору ошибок мы делаем следующее:

- воссоздаем хронологическую последовательность и собираем детали с разных точек зрения о том, как произошел сбой, проверяя, что мы при этом никак никого не наказываем;
- мотивируем всех инженеров улучшать безопасность, побуждая их давать подробные описания того, как их действия способствовали возникновению сбоя;
- поощляем людей, совершающих ошибки, становиться экспертами, показывающими остальным, как избежать этих ошибок в будущем;
- допускаем, что всегда есть ситуации, где люди могут действовать только по своему усмотрению, а оценить эти действия можно только в ретроспективе;
- предлагаем ответные меры, чтобы предотвратить похожие сбои в будущем, и проверяем, что для этих мер записана целевая дата и определен ответственный.

Чтобы лучше понять, что же на самом деле произошло, на совещании должны присутствовать следующие участники:

- принимавшие решения, имеющие отношение к проблеме;
- обнаружившие проблему;
- устранявшие проблему;
- диагностировавшие проблему;
- пострадавшие от проблемы;
- все остальные заинтересованные в разборе проблемы.

Первая задача во время совещания — восстановить цепочку событий, имевших отношение к сбою. Сюда включаются все наши действия и время их совершения (в идеале — подкрепленные логами чатов, например IRC или Slack), то, какие эффекты мы наблюдали (в идеале — в виде конкретной телеметрии, а не в виде субъективного изложения фактов), все рассмотренные нами пути исследования проблемы и предложенные способы решения.

Для такого анализа мы должны скрупулезно фиксировать детали и укреплять культуру, предполагающую возможность делиться информацией без страха наказания. Поэтому будет хорошо, особенно для первых совещаний, если встречи будет вести специально обученный посредник, никак не связанный с возникшей проблемой.

Во время совещания и последующего решения вопроса нужно запретить употребление фраз «мог бы» и «должен был», так как это *контрафактуальные высказывания*, возникающие из свойства человеческого мышления создавать альтернативные варианты уже прошедших событий.

Контрафактуальные утверждения, такие как «Я мог бы...» или «Если бы я знал об этом, я бы...», формулируют проблему в терминах *воображаемой системы*, а не в терминах *системы, существующей на самом деле*, которой мы и должны себя ограничивать (см. ).

Один из возможных неожиданных результатов таких встреч — сотрудники часто станут винить себя за то, что оставалось вне их контроля, или сомневаться в своих способностях. Йен Малпасс, инженер Etsy, замечает: «Когда мы делаем что-то такое, отчего весь сайт вдруг перестает работать, все внутристынет, как от прыжка в ледяную воду. Наверное, первая мысль у всех — “Я неудачник, я понятия не имею, что делаю”. Надо это прекратить, такие мысли — путь к отчаянию, сумасшествию и ощущению себя обманщиком. Нельзя допускать, чтобы хорошие инженеры так о себе думали. Гораздо лучший вопрос: “Почему тогда это казалось мне правильным?”»

Во время совещаний мы должны выделить достаточно времени на мозговой штурм и на выбор ответных мер. Когда контрмеры определены, нужно определить их приоритет, составить план выполнения и выбрать ответственного за их воплощение в жизнь. Все это показывает, что мы ценим улучшение повседневной работы больше, чем ее саму.

Дэн Мильстейн, один из ведущих инженеров Hubspot, пишет, что он начинает совещания по разбору ошибок со слов: «Давайте попытаемся подготовиться к такому будущему, где мы такие же тупые, как сегодня». Другими словами, контрмера «быть осторожнее» или «быть не такими глупыми» — плохая, вместо этого мы должны разработать реальные контрмеры, чтобы подобные ошибки больше не повторялись.

Примерами контрмер могут быть автоматизированные тесты для обнаружения опасных состояний в цикле развертывания, добавление новой телеметрии, определение категорий изменений, подразумевающих дополнительное рецензирование коллегами, и проведение репетиций сбоев во время регулярных упражнений игровых дней.

Проведя совещание по разбору ошибок, нужно сообщить всем о доступности записей со встречи и других документов (например, записей о восстановленном ходе событий, логов IRC-чата, внешних контактов). Эта информация в идеале должна находиться в централизованном месте, где все сотрудники компании могут получить к ней доступ и извлечь пользу и новые знания из произошедшего сбоя. Проведение совещаний с разбором ошибок настолько важно, что можно даже приостановить полное устранение инцидента до того, как будет завершен анализ сбоя.

Такой подход помогает распространять локальные улучшения и опыт по всей компании. Рэнди Шуп, бывший технический директор Google App Engine, описывает то, как документация совещаний по разбору ошибок может иметь огромную ценность для организации: «Как вы можете догадаться, в Google вся информация доступна. Все документы с разбора причин сбоев находятся в местах, где их могут видеть все сотрудники. И поверьте мне, когда у какой-то группы происходит авария, похожая на то, что уже когда-то было, эти документы читаются и изучаются в первую очередь».

Широкое распространение результатов анализа ошибок и поощрение знакомства с ними увеличивают суммарные знания компании. Кроме того, среди организаций, занимающихся онлайн-услугами, все более распространенными становятся публикации разборов инцидентов, повлиявших на клиентов. Это часто сильно увеличивает прозрачность работы компаний для внутренних и внешних клиентов и, в свою очередь, повышает доверие к нам.

Стремление проводить как можно больше совещаний по разбору ошибок привело компанию Etsy к некоторым проблемам: за четыре года в базе организации накопилось огромное число заметок со встреч. Искать информацию, сохранять новые данные и работать с базой знаний стало очень трудно.

Чтобы справиться с проблемой, в компании придумали инструмент под названием Morgue, позволяющий легко фиксировать аспекты каждого сбоя, например его MTTR и степень серьезности, лучше работать с разными часовыми поясами (это стало важно, потому что многие сотрудники Etsy начали работать удаленно) и включать в отчеты другие данные, например текст в формате Markdown, изображения, теги и историю.

Приложение Morgue было разработано для того, чтобы команде было легко фиксировать:

- возникла ли проблема из-за запланированного или незапланированного инцидента;
- кто ответствен за разбор ошибок;
- важные логи IRC-чата (особенно важно для проблем, возникших в три часа ночи, когда точное фиксирование деталей может не произойти);
- важные тикиеты JIRA для корректирующих действий и дедлайны по ним (эта информация особенно важна для менеджмента);
- ссылки на форумные посты клиентов (где клиенты жалуются на проблемы).

После разработки и использования Morgue число фиксируемых разборов в Etsy сильно увеличилось по сравнению с тем временем, когда они использовали страницы специальной вики, особенно для инцидентов P2, P3 и P4 (то есть инцидентов с низким уровнем серьезности). Этот результат подтвердил гипотезу, что если документировать разбор ошибок с помощью инструментов типа Morgue станет проще, то больше специалистов начнут записывать и детализировать результаты совещаний, и накопленный опыт организации увеличится.

Эми Эдмондсон, профессор управления и менеджмента Гарвардской школы бизнеса и соавтор книги *Building the Future: Big Teaming for Audacious Innovation*, пишет:

Путь решения проблемы, не обязательно требующий больших затрат времени и денег, — избавиться от предрассудков в отношении ошибок. Эли Лилли делает это еще с ранних 1990-х: она устраивает «вечеринки неудачников», чтобы отметить умные, высококачественные научные эксперименты, закончившиеся неудачей. Эти вечеринки обходятся недорого, а перераспределение ценных ресурсов — а именно ученых — на новые проекты раньше, чем обычно, может сэкономить сотни тысяч долларов, не говоря уже о возможных стимулах для новых открытий.

Когда организации учатся эффективно диагностировать и решать проблемы, то, чтобы не останавливаться в развитии, они неизбежно расширяют понятие того, что считать проблемой. Для этого нужно научиться усиливать слабые сигналы о возможных неполадках. Например, как было описано в части IV, к тому моменту, когда компания Alcoa смогла существенно сократить количество несчастных случаев на производстве, Пол О'Нил, CEO Alcoa, начал получать отчеты не только о реально произошедших несчастных случаях, но и о потенциально аварийных ситуациях.

Доктор Стивен Спир резюмирует достижения О'Нила, отмечая: «Хотя все началось с проблем безопасности труда, в компании быстро обнаружили, что эти проблемы отражали общее невежество в отношении процессов производства, и эта невежественность проявлялась в других проблемах, связанных с качеством, своевременностью работы и количеством брака».

Когда мы работаем в сложных системах, необходимость усиливать слабые сигналы очень важна, чтобы избежать катастрофических последствий. Прекрасный пример — как NASA использовала сигналы о неполадках в эпоху космических шаттлов. В 2003 г., на шестнадцатый день полета космического шаттла Columbia, он взорвался при входе в атмосферу. Сейчас известно, что это произошло из-за того, что во время взлета от внешнего топливного бака оторвался кусок теплоизоляционной пены.

Незадолго до входа шаттла в атмосферу несколько инженеров NASA среднего звена сообщили об этом инциденте, но их не услышали. Во время послестартового анализа инженеры заметили на видеозаписи, что отпавший кусок теплоизоляции повредил теплоизоляцию на крыле, и оповестили об этом менеджеров NASA, но им ответили, что проблемы с теплоизоляцией бывают часто. Смещение изоляционной пены повреждало шаттлы и во время предыдущих запусков, но это никогда не приводило к авариям. Считалось, что это проблема технического обслуживания, не стоит из-за нее принимать специальных мер.

Майкл Роберто, Ричард Бомер и Эми Эдмондсон в статье 2006 г. для Harvard Business Review писали, что культура NASA стала одной из причин катастрофы. Они описывали две типичные структуры организаций: *стандартизированная модель*, где всем управляют установленные практики и системы и где жестко соблюдают сроки и бюджеты, и *экспериментальная модель*, где каждый день каждую задачу и каждую крупицу новой информации оценивают и обсуждают в атмосфере, напоминающей научно-исследовательскую и конструкторскую (R&D) лабораторию.

Они также отмечают: «Фирмы сами навлекают на себя неприятности, принимая неправильный образ мышления, диктующий, как реагировать на *неоднозначные угрозы* или, в терминах этой книги, на *слабые сигналы о неполадках*... К 1970 г. NASA создала культуру с жесткой стандартизацией, рекламируя Конгрессу шаттлы как дешевый и многоразовый космический корабль». Вместо экспериментальной модели, где каждый факт должен был оцениваться без предвзятости, в NASA предпочитали четкое следование процедурам. Отсутствие непрерывного обучения и экспериментирования привело к катастрофическим последствиям. Авторы резюмируют: важны именно культура и образ мыслей, а не просто «необходимость быть осторожным», «бдительность сама по себе не может помешать неоднозначным слабым сигналам о неполадках перерасти в дорогостоящие (а иногда и трагичные) ошибки».

Работа в технологическом потоке ценности, например связанном с космическими технологиями, должна восприниматься как принципиально экспериментальное занятие, и управляться она должна соответственно. Вся сделанная работа — это новая потенциально важная гипотеза и источник данных, а не рутинное воспроизведение и подтверждение прежних методов. Вместо того чтобы считать технологическую работу полностью стандартизованной, когда все стремятся к слепому следованию установленным процедурам, нужно постоянно выявлять все более слабые сигналы о возможных сбоях, чтобы лучше понять системы и управлять ими.

Руководители компаний, сознательно или нет, своими действиями укрепляют организационную культуру и ее ценности. Эксперты по аудиту, бухгалтерскому учету и этике давно заметили, что взгляды «верхушки» предопределяют вероятность мошенничества или других недобросовестных действий. Чтобы укрепить культуру обучения и взвешенных рисков, руководители должны постоянно следить, чтобы сотрудники не боялись ошибок, но в то же время чувствовали себя ответственными за их исправление и за получение нового опыта.

Говоря об ошибках, Рой Рапопорт из Netflix замечает: «Что доклад 2014 State of DevOps Report доказал мне, так это то, что высокоеффективные DevOps-организации совершают ошибки чаще. Это не только нормально, это как раз то, что компаниям и нужно! Если высокоеффективные компании работают в 30 раз быстрее и при этом уровень сбоев в два раза ниже, очевидно же, что общее число сбоев там выше».

Далее он продолжает: «Я как-то говорил с одним коллегой о недавнем масштабном сбое у нас в Netflix, он произошел, честно говоря, из-за глупейшей ошибки. На самом деле сбой случился из-за одного инженера, за последние 18 месяцев уже дважды полностью выводившего Netflix из строя. Но, конечно, мы его ни за что не уволили бы. За те же 18 месяцев этот инженер продвинул уровень наших эксплуатации и автоматизации вперед не на километры, а на световые годы. Его работа позволила нам безопасно делать развертывания каждый день, и он сам лично провел огромное число развертываний кода».

Рапопорт заключает: «В DevOps должно быть место для таких инноваций и для вытекающего из них риска ошибок. Да, в эксплуатации у вас будет больше сбоев. Но это хорошо, за это нельзя наказывать».

Как мы видели во введении к этой главе, намеренное создание неполадок в среде эксплуатации (например, с помощью Chaos Monkey) — один из способов укрепления адаптивности к сбоям. В этой части мы опишем процессы симуляции и введения сбоев в систему, чтобы удостовериться, что наши системы спроектированы и выстроены правильно, а неполадки контролируются и происходят, как

запланировано. Для этого мы регулярно (или даже непрерывно) будем проводить тесты, чтобы убедиться: системы выходят из строя плавно и предсказуемо.

Как пишет Майкл Нейгард, автор книги *Release It! Design and Deploy Production-Ready Software*, «Подобно тому как в машины встраивают зоны смятия, чтобы при аварии пассажиры оставались в безопасности, вы можете определить, какие элементы системы важнее всего, и спроектировать режимы отказа, способные держать трещины подальше от этих элементов. Если вы специально не определяете режимы отказа, то результат будет непредсказуемым — и, как правило, опасным».

Адаптивность требует, чтобы мы сначала продумали режимы отказа и затем протестировали, работают ли они так, как нам нужно. Один из способов проделать это — специально создавать сбои в эксплуатационной среде и репетировать масштабные аварии, чтобы убедиться, что мы можем быстро восстановиться после инцидентов, когда они на самом деле произойдут, в идеале — так, чтобы клиенты этого не заметили.

История Netflix и сбоя Amazon AWS-EAST в начале этой главы — не единственный пример. Еще более интересный пример адаптивности Netflix произошел во время «Великой перезагрузки Amazon 2014», когда почти 10 % всех серверов EC2 Amazon пришлось перезагрузить, чтобы срочно установить обновление для системы безопасности Xen. Кристос Каланцис, специалист по конструированию облачных баз данных в Netflix, вспоминает: «Когда мы узнали о срочной перезагрузке EC2, у нас отвисла челюсть. Когда мы получили список того, сколько серверов Cassandra будут затронуты этим, мне стало плохо». Каланцис продолжает: «Потом я вспомнил все тренировки с Chaos Monkey. Моя реакция была: “Ну же, давайте!”»

И опять результаты оказались поразительными. Из более чем 2700 узлов Cassandra, используемых в эксплуатации, 218 были перезагружены, а перезагрузка 22 закончилась неудачей. Каланцис и Брюс Вонг, занимающийся проектированием хаоса (Chaos Engineering), писали: «В те выходные Netflix был недоступен ровно ноль секунд. Регулярные и постоянные упражнения в устранении сбоев, даже на уровне баз данных, должны быть частью плана по развитию адаптивности каждой компании. Если бы Cassandra не участвовала в тренировках Chaos Monkey, эта история закончилась бы совсем по-другому».

Что еще более удивительно, в те выходные не только в Netflix никто не работал над устранением инцидентов из-за сбоев узлов Cassandra, никого вообще не было в офисе — все сотрудники были в Голливуде, где отмечали завершение очередного этапа работ. Это еще один пример того, как благодаря проактивному фокусированию на адаптивности то, что для одних компаний могло оказаться серьезным кризисом, для других — всего лишь еще один обычный рабочий день (см. ).

В этой части мы опишем специальные тренировки восстановления после аварий. Они называются игровыми днями (Game Days). Этот термин придуман Джессом Роббинсом, одним из основателей сообщества Velocity Conference и сооснователем компании Chef, и стал популярен благодаря его работе в компании Amazon, где он отвечал за программы по обеспечению доступности сайтов и где его называли «мастером аварий». Идея игровых дней появилась из такого направления, как *адаптивное проектирование* (resilience engineering). Роббинс определяет адаптивное проектирование как «тренировку, направленную на увеличение адаптивности с помощью намеренных масштабных сбоев во всех критических системах».

Роббинс отмечает, что, «когда вы собираетесь спроектировать маштабируемую систему, лучшее, на что вы можете надеяться, — это надстроить надежную платформу над абсолютно ненадежными компонентами. Из-за этого вы оказываетесь в ситуации, где сложные сбои неизбежны и непредсказуемы».

Следовательно, мы должны обеспечить бесперебойную работу сервисов во время аварий, теоретически во всей системе, в идеале — без кризисного вмешательства (или вручную). Как шутит Роббинс, «сервис протестирован только тогда, когда мы ломаем его в эксплуатации».

Цель Game Days — помочь командам симулировать и репетировать сбои, чтобы у них была возможность практиковаться. Сначала мы планируем катастрофическое событие, например разрушение целого data-центра. Потом мы даем командам время на подготовку, устранение всех возможных точек отказа и создание необходимых процедур наблюдения, перехода на другие ресурсы и так далее.

Команда Game Day определяет и проводит тестовые задания, например проверяет переключение баз данных (то есть симулирует отказ первичной базы данных и переключение на вторую базу) или отключает важное сетевое соединение, чтобы выявить проблемы в каком-либо из анализируемых процессов. Все обнаруженные проблемы и сложности анализируются, устраняются и снова тестируются.

В запланированный момент мы устраиваем сбой. Как это описывает Роббинс, в Amazon они «буквально отключали питание устройств — без предупреждения — и затем позволяли системам

выходить из строя естественным образом, а сотрудникам — следить за этими процессами, независимо от того, как они бы стали развиваться».

Такой подход помогает нам выявлять *скрытые дефекты* наших систем. Они оказываются доступны для наблюдения только благодаря намеренному внесению сбоев в системы. Роббинс объясняет: «Вы можете обнаружить, что некоторые системы наблюдения или управления, нужные для процессов восстановления, не работают из-за той же симулированной вами аварии. Или вы можете найти новые точки отказа, о которых вы до этого и не подозревали». Такие тренировки постепенно становятся все более интенсивными и сложными с той целью, чтобы они воспринимались как обычная часть обычного рабочего дня.

С помощью игровых дней мы постепенно формируем более адаптивные сервисы, получаем уверенность в том, что мы можем восстановить работу после инцидентов, а также создаем новые знания и повышаем способность к адаптации нашей компании.

Отличный пример симулирования сбоев — программа восстановления после аварий компании Google (Disaster Recovery Program, DiRT). Крипа Кришнан, главный инженер программ Google, на момент написания этой книги руководит этой программой уже семь лет. За это время они симулировали землетрясение в Кремниевой долине, из-за которого комплекс зданий и офисов Google в городе Маунтин-Вью оказался отсоединен от остальной компании, полную потерю питания в крупнейших data-центрах и даже нападение пришельцев на города, где жили инженеры организации.

По словам Кришнан, «тестировщики часто обходят стороной бизнес-процессы и коммуникации. Системы и процессы очень тесно переплетены, и разделять тестирование систем и тестирование бизнес-процессов — нереалистичный подход: отказ бизнес-системы скажется на бизнес-процессе, и наоборот, работающая система без нужного персонала не очень-то полезна».

Во время симулирования таких аварий было сделано несколько открытий:

- когда соединение было прервано, переход коммуникации на рабочие места инженеров не помог;
- инженеры не знали, как получить доступ к коммутатору телеконференции, или коммутатор мог соединять только пятьдесят человек, или им нужен был новый провайдер конференций, позволяющий выбрасывать из беседы участников, не бравших трубку и вынуждающих всех остальных слушать мелодию ожидания ответа;
- когда у data-центров закончилось топливо для запасных генераторов, никто не знал процедур для экстренных закупок у поставщика, из-за чего одному сотруднику пришлось использовать личную кредитную карту и закупить топлива на 50 000 долларов.

С помощью создания аварий в контролируемых условиях мы можем успешно тренироваться и придумывать нужные сценарии. Еще один важный результат Game Days — то, что работники знают, кому звонить и с кем разговаривать. Так они налаживают отношения с сотрудниками других отделов, чтобы можно было успешно работать вместе во время аварий, превращая сознательные действия в бессознательные шаблоны и привычки.

Чтобы создать справедливую культуру, поощряющую обучение, нам нужно поменять отношение к так называемым ошибкам. При правильном подходе ошибки, неизбежные в сложных системах, создают динамическую учебную среду, где все сотрудники чувствуют себя защищенными и могут выдвигать новые идеи и замечания и где команды быстрее оправляются от неудачных проектов, работавших не так, как ожидалось.

Разбор ошибок без поиска виноватых и сознательное создание сбоев укрепляют культуру, где всем комфортно и где все чувствуют ответственность за получение новых знаний из ошибок. Кроме того, когда мы значительно сокращаем число инцидентов, мы уменьшаем порог чувствительности, чтобы не останавливаться в развитии. Как говорит Питер Сэндж, «единственное надежное конкурентное преимущество — это способность компании учиться быстрее, чем ее конкуренты».

## Глава 20. Преобразуйте локальные открытия в глобальные улучшения

В предыдущей главе мы обсудили, как с помощью разбора ошибок без поиска виноватых побуждать исполнителей говорить о своих ошибках и тем самым создавать безопасную и ориентированную на обучение культуру. Мы также изучили то, как находить слабые сигналы о возможных сбоях, а также побуждать сотрудников экспериментировать и рисковать. Кроме того, с помощью проактивного планирования и тестирования возможных аварий, а также поиска и исправления скрытых дефектов мы сделали наши системы более адаптивными и безопасными.

В этой главе мы создадим механизмы, помогающие распространить новые знания и улучшения по всей организации, тем самым многократно увеличивая эффект новых открытий. Благодаря этому все сотрудники смогут извлечь пользу из накопленного опыта компании, и мы выведем практические достижения нашей организации на новый уровень.

Во многих организациях есть чаты, упрощающие быстрое общение между командами. Однако чаты также используются и для автоматизации.

Эта методика впервые была опробована в GitHub, где она стала известна под названием ChatOps. Главной целью было использовать автоматизированные инструменты прямо в разговорах чатов, создавая прозрачность и документируя текущую работу. Как пишет Джесс Ньюлэнд, системный инженер GitHub, «Даже если вы новичок, вы можете посмотреть на логи чатов и увидеть, как все было сделано. Вы как будто все время занимались парным программированием со всей командой».

Они создали Hubot, приложение, взаимодействовавшее с командой эксплуатации в их чатах, где можно было выполнять действия с помощью команд прямо из беседы (например, @hubot deploy owl to production). Результаты выполнения также отображались в чате.

У автоматического выполнения команд из чата (в противоположность запуску автоматизированных скриптов из командной строки) было много преимуществ, в том числе:

- все видят всё, что происходит;
- новые инженеры в первый же рабочий день могли видеть, как выглядит повседневная работа и как ее выполняют другие;
- сотрудники чаще просили о помощи, когда видели, что остальные помогают друг другу;
- новые знания внутри компании накапливались гораздо быстрее.

Кроме того, помимо этих преимуществ, чаты записывают все разговоры и делают обсуждения общедоступными. Электронная почта же обычно личная, информацию оттуда узнать или распространить по компании нелегко.

Автоматизация в чатах помогает документировать и делиться нашими наблюдениями и способами решения проблем прямо в процессе работы. Это укрепляет культуру прозрачности и сотрудничества в нашей повседневной деятельности.

Это также очень эффективный способ распространения знаний по компании. В GitHub все сотрудники, занимавшиеся эксплуатацией, работали удаленно. Более того, все инженеры эксплуатации жили в разных городах. Как вспоминает Марк Имбриако, бывший директор по эксплуатации GitHub, «в GitHub не было материального кулера. Кулером был чат».

Через Hubot можно было запускать разные инструменты автоматизации, используемые в GitHub, такие как Puppet, Capistrano, Jenkins, resque (библиотека на основе Redis для создания фоновых процессов) и graphme (инструмент для создания графиков в Graphite).

С помощью Hubot можно было проверять работоспособность сервисов, развертывать код в производственную среду и блокировать оповещения, когда сервисы переходили в профилактический режим. Повторяющиеся действия, например smoke-test при сбое развертывания, выведение серверов эксплуатации из работы, возврат к исходной ветке для front-end-сервисов или извинения инженерам, вызванным в нерабочее время, тоже стали возможными действиями в Hubot.

Результаты внедрения изменений в репозиторий исходного кода и команды, запускавшие процесс развертывания, появлялись в чате. Кроме того, когда внесенные изменения продвигались по конвейеру развертывания, их статус также отображался в чате.

Типичный разговор в чате мог выглядеть так:

«@sr: @jnewland, как получить список больших репозиториев? что-то вроде disk\_hogs?»

«@jnewland:/disk\_hogs»

Ньюлэнд отмечает: некоторые вопросы, которые раньше часто задавали во время работы над проектом, теперь звучат очень редко. Например, инженеры могут спрашивать друг у друга: «Как там развертывание?», или «Ты проведешь развертывание или я?», или «В каком состоянии загрузка?»

Среди всех преимуществ — в их числе быстрая адаптация новых сотрудников и повышение производительности — Ньюлэнд особо выделяет то, что работа стала более человечной, инженеры теперь могли быстро и без усилий находить проблемы и помогать друг другу в их решении.

GitHub создал среду для совместного поиска новых знаний, которые могут быть трансформированы в статус организационных. Делиться знаниями с коллегами стало очень легко. Далее в этой главе мы рассмотрим способы того, как создавать пути и ускорять распространение нового опыта.

Мы слишком часто закрепляем стандарты и процессы проектирования архитектуры, тестирования, развертывания и управления инфраструктурой в документах Word. Затем они пылятся в дальнем углу какого-нибудь сервера. Проблема в том, что инженеры, разрабатывающие новые приложения или среды, не всегда знают, что эти документы существуют, или у них нет времени реализовывать требования стандартов. В результате они создают свои инструменты и процессы с ожидаемым исходом: хрупкие, неподдерживаемые и ненадежные приложения, а также среды, которые сложно использовать, поддерживать и развивать.

Вместо того чтобы записывать опыт в файлы Word, мы должны так преобразовать эти стандарты и процессы, содержащие накопленный опыт компании, чтобы их было легко использовать. Один из лучших способов сделать эти знания полезными — поместить их в централизованный репозиторий и сделать его доступным для всех сотрудников.

Джастин Арбакл, будучи главным архитектором компании GE Capital, рассказывал: «Нам нужно было создать механизм, позволяющий командам легко соблюдать требования: государственные, региональные, отраслевые, описанные в десятках нормативных баз, распространяющиеся на тысячи приложений, работающих на десятках тысяч серверов в десятках данных центров».

Созданный механизм получил название ArchOps. Этот механизм «позволял нашим инженерам быть строителями, а не каменщиками. Когда мы перевели стандарты проектирования в автоматизированные шаблоны, простые в использовании даже для новичка, консистентность оказалась побочным продуктом нашей работы».

Трансформируя процессы, осуществляемые вручную, в автоматизированный выполняемый код, мы делаем их доступными и полезными для всех, кто их использует. Арбакл заключает: «Уровень соблюдения требований в компании прямо пропорционален степени того, насколько эти требования переведены в код».

Если автоматизированный процесс — самый удобный способ достичь цели, то его будут использовать чаще и чаще, и можно даже подумать о том, чтобы сделать его общекорпоративным стандартом.

Общедоступный репозиторий исходного кода для всей компании — один из самых мощных механизмов распространения опыта по организации. Когда мы обновляем что-либо в таком репозитории (например, библиотеку общего пользования), изменения быстро и автоматически распространяются на все сервисы, использующие эту библиотеку.

Компания Google — один из самых ярких примеров использования репозитория, доступного всей организации. К 2015 г. в нем хранилось более миллиарда файлов и свыше двух миллиардов строк кода. Это хранилище кода используется каждым из 25 000 инженеров компании и покрывает все ее сервисы, включая Google Search, Google Maps, Google Docs, Google+, Google Calendar, Gmail и YouTube.

Один из ценных результатов такого подхода — то, что инженеры могут использовать разнообразный опыт всех сотрудников компании. Рэйчел Потвин, технический руководитель, возглавляющий группу по организации инфраструктуры для разработчиков, в интервью для журнала Wired рассказала, что у каждого инженера Google есть доступ к огромному объему библиотек, где «практически все уже давно сделано до них».

Кроме того, как поясняет Эран Мессери, инженер команды по организации инфраструктуры для разработчиков в Google (Google Developer Infrastructure group), одно из преимуществ использования единого хранилища в том, что все пользователи имеют доступ к самой последней версии кода и ничего не нужно координировать.

Помимо исходного кода в едином репозитории можно хранить и другие вещи, фиксирующие знания и опыт компании, например:

- конфигурационные стандарты для библиотек, инфраструктуры и сред (рецепты Chef, декларации Puppet и так далее);
- инструменты развертывания;
- стандарты и инструменты тестирования, включая стандарты безопасности;
- инструменты конвейера развертывания;
- инструменты мониторинга и анализа;
- руководства и стандарты.

Сохранение опыта и создание общего доступа через такое хранилище — один из самых мощных механизмов распространения знаний. Как пишет Рэнди Шуп, «самый мощный механизм для предотвращения сбоев в Google — это единый репозиторий. Когда кто-то отправляет туда свой код, новая сборка всегда будет использовать последние версии всего. Все собирается напрямую из исходников вместо того, чтобы динамически подключать во время выполнения необходимые компоненты. В каждый момент времени существует лишь одна версия библиотеки, она и подключается во время сборки приложения».

Том Лимончелли — соавтор книги *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems* и бывший инженер SRE Google. В своей книге он утверждает, что ценность единого репозитория настолько высока, что это сложно выразить словами.

«Вы можете написать какой-нибудь инструмент всего один раз, и он будет использоваться во всех проектах. У вас есть стопроцентно точное знание о том, кто зависит от этой библиотеки; поэтому вы можете провести рефакторинг и быть полностью уверенным в том, кого затронут изменения и кому нужно провести дополнительные тесты. Наверное, я мог бы привести еще сотню примеров. Я не могу выразить словами, насколько это важное конкурентное преимущество для Google».

В Google у каждой библиотеки (например, libc, OpenSSL, а также библиотеки, разработанные самой компанией, например для поддержания многопоточности в Java) есть свой хранитель, ответственный за то, чтобы она не только компилировалась, но и успешно проходила тесты для всех зависящих от нее проектов, совсем как настоящий библиотекарь. Хранитель также отвечает за перевод каждого проекта со старой версии на новую.

Представьте реальную организацию, пользующуюся 81 версией библиотеки Java Struts. Всех версий, кроме одной, серьезно уязвимы. Поддержка всех этих вариантов — у каждого свои особенности — создает огромную операционную нагрузку. Кроме того, такое большое количество делает обновление версий рискованным и опасным, из-за чего разработчики не горят желанием эти обновления проводить. Порочный круг.

Единое хранилище исходного кода решает большинство этих проблем, и к тому же можно создать автоматизированные тесты, позволяющие командам переходить на новые версии уверенно и безопасно.

Если мы не можем собирать все приложения с единого дерева исходного кода, нужно найти другой способ поддержки хороших версий библиотек и их зависимостей. Например, можно завести единое хранилище, такое как Nexus, Artifactory или репозиторий Debian или RPM. Это хранилище затем необходимо регулярно обновлять, если в репозиториях или в производственных системах будут выявлены уязвимые места.

После того как мы создали общие для всей организации библиотеки, стоит заняться быстрым распространением профессиональных компетенций и улучшений. Включение в библиотеки большого количества автоматизированных тестов сделает их самодокументирующими, и другим инженерам будет легко понять, как они работают.

Если мы внедрили методологию разработки через тестирование (TDD), где тесты пишутся до кода, то такое преимущество мы получим практически без лишних действий. Такой подход превращает наши наборы тестов в актуальную спецификацию системы. Любой сотрудник, желающий понять, как работать с какой-либо системой, может просто взглянуть на набор тестов и увидеть работающие примеры того, как пользоваться API-системой.

В идеале у каждой библиотеки должен быть свой хранитель или своя команда. Они становятся источником знаний о данной библиотеке. Кроме того, в эксплуатации должна использоваться только одна, наилучшая версия библиотеки (в идеальном случае), отражающая накопленные знания и опыт организации.

В этой модели инженер, ответственный за библиотеку, также отвечает и за перевод всех групп, пользующихся его репозиторием, с устаревшей версии на новую. Это, в свою очередь, требует быстрого обнаружения ошибок перехода между версиями библиотеки. Помочь могут исчерпывающее автоматизированное тестирование и непрерывная интеграция для всех систем, использующих ту или иную библиотеку.

Чтобы быстрее распространять знания, можно также создать группы обсуждений или чаты для каждой библиотеки и сервиса. Благодаря этому любой сотрудник, имеющий вопросы, может получить ответ от других пользователей, часто отвечающих быстрее, чем разработчики.

С помощью такого инструмента коммуникации мы облегчаем обмен знаниями и опытом, а сотрудники получают возможность помогать друг другу с проблемами и новыми подходами к разработке.

Когда разработчики следят за судьбой своего продукта после того, как закончили его создание, и участвуют в ликвидации сбоев в производственной среде, сопровождать приложение становится гораздо проще и безопаснее. Кроме того, когда приложения и код сознательно пишутся с расчетом на высокую скорость потока и быстрые развертывания, то появляется возможность сформулировать набор нефункциональных требований, заслуживающих интеграции во все сервисы компании.

Благодаря таким нефункциональным требованиям наши сервисы будет легко развертывать и поддерживать, замечать и решать проблемы будет проще, а при отказе неисправных компонентов система не будет полностью выходить из строя. Примерами нефункциональных требований могут быть:

- достаточный объем производственной телеметрии в приложениях и средах;
- способность четко отслеживать зависимости;
- адаптивные сервисы, способные поддерживать минимум функциональности даже при масштабных сбоях;
- прямая и обратная совместимость между версиями;
- способность архивировать данные для управления размером набора данных в эксплуатации;
- легкий поиск и интерпретация логов всех сервисов;
- способность отслеживать запросы пользователей через большое количество сервисов;
- простая централизованная конфигурация вычислительной среды с флагками элементов функциональности и так далее.

При определении таких типов нефункциональных требований использовать накопленный опыт компании в новых и уже существующих сервисах становится гораздо проще. Ответственность за это лежит на команде, разрабатывающей тот или иной сервис.

Если в эксплуатацию введена работа, не поддающаяся полной автоматизации или неспособная перейти в режим самообслуживания, то ее надо сделать настолько шаблонной и воспроизводимой, насколько возможно. Для этого необходимо стандартизировать эту работу, максимально автоматизировать и тщательно задокументировать весь процесс, чтобы другим командам было проще планировать и выполнять такие действия.

Вместо того чтобы собирать серверы вручную и только потом вводить их в производственную среду в соответствии с чек-листами, нужно автоматизировать эту работу, насколько возможно. Если некоторые действия все равно требуют вмешательства вручную (например, установка и подключение серверов, проводимые другой командой), то нужно четко определить момент перехода задачи к другой группе, чтобы сократить времяостоя и уменьшить число ошибок. Все это также поможет планировать такие шаги в будущем. Например, для автоматизации и организации рабочего процесса можно использовать инструменты Rundeck или системы тикетов вроде JIRA или ServiceNow.

В идеале для всех повторяющихся действий в эксплуатации мы должны знать следующее: какое действие нужно произвести, кто для этого нужен, какие шаги требуются для его выполнения и так далее. К примеру, «мы знаем, что релиз стабильной окончательной версии происходит в четырнадцать этапов, требует привлечения четырех команд и последние пять раз в среднем на это требовалось три дня».

Точно так же, как мы реализуем требования заказчиков в разработке, мы можем создать четко определенные «требования инженеров эксплуатации». В них отражаются действия, повторяющиеся во всех проектах (например, развертывание, вопросы безопасности и производительности и так далее). С помощью таких требований мы согласовываем работу отделов разработки и эксплуатации и упрощаем планирование, а также получаем на выходе более стабильные результаты.

Когда одна из наших целей — максимизировать продуктивность разработчиков, а архитектура у нас — сервис-ориентированная, то небольшие команды могут писать свои приложения на наиболее подходящем для них языке и в оптимальной среде разработки. В некоторых случаях такой подход лучше всего согласуется с целями компании.

Однако есть ситуации, где происходит обратное, например когда поддержка важного сервиса лежит на одной команде и только она может устранять неполадки и вносить изменения. В результате в системе появляется узкое место. Другими словами, продуктивность команды оптимизирована, но при этом оказалась в противоречии с целями компании.

Такое часто происходит, когда за все аспекты поддержки сервиса отвечает только одна функционально ориентированная группа по эксплуатации. В такой ситуации нужно сделать так, чтобы отдел эксплуатации мог влиять на то, какие компоненты используются в производственной среде, или же снять с них ответственность за неподдерживаемые платформы.

Если у нас нет готового списка поддерживаемых в эксплуатации технологий, составленного при участии отделов разработки и эксплуатации, нужно пройтись по инфраструктуре производственной среды и задействованным в ней сервисам, а также по их текущим зависимостям, чтобы найти компоненты, создающие непропорционально большой риск сбоев или незапланированных работ. Наша цель — определить технологии, которые:

- замедляют или мешают рабочему процессу;
- являются причиной большого количества незапланированной работы;
- являются причиной большого количества запросов на поддержку;
- плохо соответствуют целям в плане характеристик архитектуры (например, производительность, стабильность, безопасность, надежность, бесперебойность работы).

Выводя проблемную инфраструктуру и платформы из сферы ответственности отдела эксплуатации, мы позволяем ему сосредоточиться на технологиях, способствующих достижению целей компании.

Как пишет Том Лимончелли, «когда я работал в Google, у нас был один официальный компилируемый язык, один официальный язык сценариев и один официальный язык пользовательского интерфейса. Да, другие языки тоже так или иначе поддерживались, но, если вы работали с “большой тройкой”, вам было гораздо проще найти нужные библиотеки, инструменты и желающих работать с вами коллег». Эти стандарты укреплялись правилами рецензирования кода, а также тем, какие языки поддерживались внутренними платформами.

Ральф Лура, директор по информационным технологиям компании Hewlett-Packard, в презентации на конференции 2015 DevOps Enterprise Summer, подготовленной совместно с Оливье Жаком и Рафаэлем Гарсиа, отмечал:

«Между собой мы описывали нашу цель как создание “буйков, а не границ”. Вместо того чтобы проводить жесткие границы, за которые никто не должен заступать, с помощью буйков мы обозначаем глубокие места канала, где вы в безопасности и где вас поддержат. Вы можете выплыть за буйки в том случае, если вы продолжаете следовать принципам организации. В конце концов, как мы создадим новую инновацию, помогающую достичь успеха, если мы не исследуем неизвестное и не работаем на самой границе? Как лидеры мы должны размечать фарватер, способствовать судоходству и позволять “матросам” исследовать то, что лежит за пределами обитаемых земель».

Во многих компаниях, переходящих на DevOps, разработчики часто рассказывают, что «инженеры эксплуатации не предоставили нам того, что мы просили, поэтому нужные нам системы мы создали и поддерживали сами». Однако в компании Etsy на ранних стадиях трансформации

руководство поступило наоборот, сильно сократив число технологий, поддерживаемых в эксплуатации.

В 2010 г., после провальных результатов пиковой нагрузки, приходящейся на рождественские праздники, команда Etsy решила значительно уменьшить количество технологий, использовавшихся в эксплуатации, выбрав те, которые могли поддерживаться во всей организации, и избавившись от всех остальных.

Их целью было стандартизировать и сознательно уменьшить инфраструктуру и число конфигураций. Одним из первых было принято решение перевести всю среду Etsy на PHP и MySQL. Это было скорее философское решение, чем технологическое — в компании хотели, чтобы и в разработке, и в эксплуатации разбирались во всем стеке технологий, чтобы все могли работать в одной среде, а также читать, переписывать и исправлять чужой код. За несколько лет, как вспоминает Майкл Рембетси, тогдашний директор по эксплуатации, «мы отправили на пенсию несколько отличных технологий, полностью выведя их из эксплуатации», в том числе lighttp, Postgres, MongoDB, Scala, CoffeeScript, Python и многие другие.

Дэн Маккинли, разработчик, первым начавший использовать MongoDB в Etsy, пишет в своем блоге, что все преимущества бессхемной базы данных перечеркивались проблемами ее эксплуатации. Здесь были и проблемы логирования, составления графиков, мониторинга, производственной телеметрии, создания резервных копий данных и их восстановления, а также многие другие проблемы, о которых разработчики обычно не должны думать. В итоге от MongoDB в компании полностью отказались и все новые сервисы создавались на основе поддерживаемой инфраструктуры баз данных MySQL.

Методики, описанные в этой главе, позволяют включить любой новый опыт в коллективное хранилище знаний организации, приумножая его положительный эффект. Это достигается благодаря активному и широкому распространению новых знаний, например с помощью чатов, подхода к архитектуре как к коду, общих репозиториев, стандартизации технологий и так далее. Благодаря такому подходу мы улучшаем практические достижения не только в разработке и в эксплуатации, но и во всей компании, потому что теперь у всех сотрудников есть доступ ко всему накопленному опыту организации.

## **Глава 21. Выделите время для обучения и улучшений**

Одна из методик, входящих в производственную систему Toyota, носит название **блиц-обучения** (или иногда *kaizen blitz*): это короткий промежуток времени, полностью посвященный решению одной конкретной проблемы, иногда длиной в несколько дней. Спир объясняет: «...блицы часто проходят так: собирается группа из нескольких человек, и они концентрируются на каком-нибудь проблемном процессе... Штурм длится несколько дней, его цель — улучшение процесса, средство для достижения цели — помочь коллег вне процесса коллегам внутри него».

Спир отмечает: часто результат блиц-обучения — возникновение нового подхода к решению проблемы. Например, появляется новая схема размещения оборудования, новый способ передачи информации, лучше организованное рабочее пространство или стандартизация деятельности. Также итогом иногда бывает список возможных изменений: их стоит ввести в обозримом будущем.

Пример блиц-обучения — программа ежемесячного испытания в додзё DevOps компании Target. Благодаря Россу Клэнтону, директору по эксплуатации Target, переход на систему DevOps прошел быстро. Одним из основных механизмов этого был центр технологических инноваций, более известный как додзё DevOps.

Додзё DevOps занимает площадь примерно в 1700 квадратных метров офисного пространства, где тренеры DevOps помогают командам улучшать свои навыки. Самый интенсивный формат называется «тридцатидневными испытаниями». За это время команды разработки работают в додзё с профильными коучами и инженерами. Команда формулирует проблему, с которой долгое время пыталась бороться, цель испытания — за 30 дней совершив прорыв в решении.

Весь месяц группа интенсивно работает с тренерами над решением выбранной задачи, каждые два дня составляя план, выполняя запланированную работу и фиксируя промежуточные результаты. Когда 30-дневное испытание завершается, члены группы возвращаются к обычной работе не только с решением важной проблемы, но и с новыми знаниями. Ими они могут поделиться со своей командой.

Клэнтон пишет: «На данный момент тридцатидневные испытания могут проводить не более восьми команд одновременно, поэтому мы фокусируемся на самых стратегически важных проектах компании. Через додзё уже прошли самые важные подразделения, включая группы, занимающиеся POS-терминалами, товарно-материальными запасами, ценообразованием и рекламными кампаниями».

Благодаря тому что участие в испытании занимает все рабочее время и участники концентрируются только на одной задаче, найденные решения часто оказываются просто поразительными.

Рави Пандей, менеджер по разработке Target, прошедший через эту программу, поясняет: «В прежние времена, чтобы получить тестовую среду, нам приходилось ждать шесть недель. Теперь же мы получаем ее за минуты и работаем рука об руку с инженерами эксплуатации. Они помогают нам действовать более продуктивно и создают нужные нам инструменты, чтобы нам было проще достигать поставленных целей. — Клэнтон продолжает: — Не так уж редко команды за несколько дней решают проблемы, которые раньше заняли бы у них от трех до шести месяцев. На данный момент через додзё уже прошло примерно 200 сотрудников. В общем они завершили 14 тридцатидневных испытаний».

В додзё также есть менее интенсивные модели работы, например Flash Build, где команды собираются на один-три дня, чтобы разработать минимально жизнеспособный продукт (*minimal viable product, MVP*) или минимально жизнеспособную функциональность (*minimal viable capability, MVC*). Кроме того, в компании каждые две недели проводят дни открытых дверей: все желающие могут прийти в додзё, чтобы поговорить с тренерами, посмотреть на результаты работы команд или пройти тренинг.

В этой главе мы опишем эти и другие способы, как выделять время на обучение и улучшения, делая их частью повседневной работы.

В этой части мы опишем ритуалы, помогающие разработчикам и инженерам эксплуатации выделять время на улучшение текущей работы, например на автоматизацию, введение нефункциональных требований и так далее. Один из самых простых способов — запланировать и регулярно выполнять блиц-обучение длиной в один день или одну неделю, когда вся команда (или вся организация) занимается наиболее волнующими проблемами и никакая работа над компонентами функциональности не разрешается. Это может быть проблемный участок кода, среда, архитектура, инструмент и так далее. Для блиц-обучения команды могут собираться из представителей всей производственной цепочки, например из разработки, ИТ-эксплуатации и службы информационной безопасности. Тех, кто обычно не работает вместе, объединяют навыки и

усилия, чтобы улучшить работу в выбранной области и затем продемонстрировать результаты всей компании.

Помимо таких терминов бережливого производства, как kaizen-blitz и блиц-обучение, методика ритуалов по улучшению работы также называется весенней или осенней генеральной уборкой (spring или fall cleaning) или неделями инверсии очереди задач (ticket queue inversion weeks). Иногда используются и другие термины, например хакатон (hack day, hackathon) или 20 % времени на инновации (20 % innovation time). К сожалению, конкретные ритуалы часто предназначены для совершенствования готовых продуктов и создания прототипов новых продуктов, а не для улучшения повседневной работы и, что еще хуже, методики используют только разработчики, что очень далеко от целей блиц-обучения.

Во время блиц-обучения целью должно быть не просто экспериментирование ради проверки новых технологий, а стремление улучшить повседневную работу. Хотя эксперименты и могут приводить к таким улучшениям, все же суть состоит в том, чтобы сконцентрироваться на одной конкретной каждойдневной проблеме.

Мы можем запланировать недельные блиц-обучения, во время которых разработка и эксплуатация работают вместе над какой-то одной задачей. Управлять таким мероприятием легко: выбирается одна неделя, и все в технологической цепочке вместе работают над одной и той же проблемой. В конце периода каждая команда устраивает презентацию коллегам и рассказывает о задаче и об итоговом решении. Такая методика укрепляет культуру: инженеры работают над проблемами во всем потоке создания ценности. Кроме того, этот подход поощряет решение проблем в ходе повседневной работы и показывает, что мы ответственно подходим к важным, но не срочным делам.

Сила блиц-обучений в том, что они вдохновляют непрерывно выявлять и решать проблемы без посторонних стимулов. Представьте, что наша сложная система — это паутина, переплетающиеся в ней нити постоянно ослабевают и рвутся. Если порвется определенная комбинация отдельных нитей, то всей паутине придет конец. Нет управления сверху, чтобы указывать рабочим, какие нити и в каком порядке нужно чинить. Вместо этого нужно создать организационную культуру и нормы, побуждающие сотрудников самим находить и исправлять слабые места системы в процессе ежедневной работы. Как отмечает Спир, «неудивительно, что пауки заделывают разрывы в паутине, как только они появляются, а не ждут, когда их станет слишком много».

Хороший пример успешного воплощения блиц-обучений описан Марком Цукербергом, генеральным директором Facebook. В интервью Джессике Стиллман, размещенном на сайте , он рассказывает: «Каждые несколько месяцев мы проводим хакатон, где все желающие создают прототипы для новых идей. В конце мероприятия вся команда собирается вместе и смотрит, что получилось. Многие из наших самых успешных продуктов появились именно во время хакатонов, в том числе Timeline, чат, видео, среда разработки для мобильных приложений и некоторые компоненты инфраструктуры, например компилятор HipHop».

Компилятор PHP HipHop особенно интересен. В 2008 г. у Facebook появились серьезные проблемы с работоспособностью: число активных пользователей превысило отметку в 100 миллионов и продолжало расти, из-за чего у технических служб были большие проблемы. Во время хакатона Хайпин Жао, старший инженер по работе с серверами в Facebook, начал экспериментировать с конвертацией кода на PHP в компилируемый код на C++, надеясь увеличить работоспособность их инфраструктуры. За следующие два года небольшая команда создала то, что потом будет названо компилятором HipHop, и перевела все сервисы Facebook с интерпретируемого PHP-кода в компилируемые двоичные файлы на C++. Новый компилятор позволил платформе Facebook выдерживать в шесть раз большие нагрузки, чем раньше.

В интервью с Кейдом Метцем, журналистом издания Wired, Дрю Пароски, один из инженеров, работавших над проектом, отметил: «Был такой момент, когда, если бы не HipHop, мы оказались бы в очень сложной ситуации. Чтобы обслуживать сайт, нам понадобилось бы больше машин, чем имелось на тот момент. Это был отчаянный шаг, но он сработал».

Позже Пароски и его коллеги Кейт Адамс и Джейсон Эванс решили, что могут улучшить производительность компилятора HipHop и убрать некоторые ограничения, снижавшие производительность разработчиков. В результате появился проект виртуальной машины HipHop (HHVM), использовавшей принцип динамической компиляции. К 2012 г. HHVM полностью заменила HipHop в эксплуатации. В разработке проекта приняло участие около 20 инженеров.

Регулярно проводя блиц-обучения и хакатоны, мы помогаем всем сотрудникам в потоке создания ценности создать что-то такое, чем они могли бы гордиться. И мы непрерывно интегрируем улучшения в нашу систему, делая ее еще более безопасной, надежной и способствующей обучению.

Динамическая культура, ориентированная на обучение, создает такие условия, в которых все могут не только учиться сами, но и учить других, используя при этом как традиционные подходы (например, занятия, тренинги), так и практические (конференции, семинары, наставничество).

Один из способов содействовать развитию этих методик — выделить для них специальное время.

Стив Фарли, директор по информационным технологиям компании Nationwide Insurance, отмечал: «У нас 5000 профессионалов. Мы называем их партнерами. С 2011 г. мы работаем над созданием культуры обучения. Ее часть — так называемые учебные четверги: каждую неделю мы выделяем время для того, чтобы наши партнеры могли учиться. В течение двух часов каждый партнер должен или учиться, или учить. Темы — все что угодно, о чем хотят узнать партнеры: о технологиях, о новинках в разработке, или о методиках улучшения процессов, или даже о том, как строить карьеру. Самое ценное, что может сделать партнер, — научить кого-то тому, что сам умеет, или научиться чему-то новому».

Как было неоднократно показано, некоторые навыки становятся все более нужными не только разработчикам, но и всем остальным инженерам. Например, инженеры эксплуатации и тестировщики должны быть знакомы с навыками, ритуалами и методиками разработки, с системами контроля версий, автоматизированным тестированием, конвейером развертывания, управлением конфигурациями и автоматизированием процессов. Знакомство с техниками разработки помогает инженерам эксплуатации быть в курсе текущих изменений, когда компания внедряет принципы и методики DevOps.

Перспектива изучения чего-то нового может быть пугающей или вызывать тревогу и стыд. Так быть не должно. В конце концов, мы учимся всю жизнь, и один из лучших способов делать это — учиться у коллег. Картик Гэквад, участвовавший в переходе компании National Instruments на принципы DevOps, сказал: «Для тех, кто работает в эксплуатации, погружение в автоматизацию не должно быть чем-то страшным — просто попросите знакомого разработчика, он будет рад помочь вам».

Мы также можем обучить людей новым навыкам, если коллеги будут наблюдать, как проходит рецензирование кода. Еще один пример — если инженеры разработки и эксплуатации работают вместе над решением небольших проблем. Так, разработчики могут показать инженерам эксплуатации, как аутентифицировать приложение, входить в систему и выполнять автоматизированные тесты, чтобы проверить работоспособность важных компонентов (ключевого функционала приложения, транзакций в базах данных, очереди сообщений). Затем можно интегрировать новые автоматизированные тесты в конвейер развертывания и время от времени запускать их, посыпая результаты в системы наблюдения и оповещения, чтобы вовремя заметить возможные сбои.

Гленн О’Доннелл, работающий в компании Forrester Research, в презентации на конференции DevOps Enterprise Summit в 2014 г. колко заметил: «Поскольку все профессионалы, любящие инновации, также любят и перемены, нас ждет прекрасное, яркое, но бедное будущее безработных, уволенных после внедрения инноваций».

Во многих организациях, сосредоточенных на контроле операционных расходов, часто препятствуют тому, чтобы инженеры посещали конференции и учились у коллег. Чтобы создать культуру обучения, мы должны поощрять посещение конференций (это касается инженеров и разработки, и эксплуатации), участие в них и, если нужно, организацию внутренних или внешних учебных мероприятий.

На данный момент DevOpsDays предлагает одну из самых захватывающих серий конференций. На этих мероприятиях рассказывается о тонкостях многих новых методик DevOps. Участие в них бесплатно или почти бесплатно, а проведение поддерживается активным сообществом профессионалов, регулярно применяющих методики на практике.

Конференция DevOps Enterprise Summit впервые состоялась в 2014 г., чтобы технические руководители могли делиться своим опытом внедрения и применения принципов DevOps в крупных организациях. Программа мероприятия строится на основе презентаций руководителей о своем опыте работы с DevOps, а также на основе докладов экспертов в разных областях на выбранные сообществом темы.

Наряду с участием во внешних конференциях многие компании, в том числе описанные в этой секции, проводят внутренние конференции для своих сотрудников.

Организация Nationwide Insurance — ведущий поставщик финансовых и страховых услуг, работающий в высокорегулируемых сферах. Среди ее предложений — страхование автомобилей и жилья. Кроме того, компания — один из крупнейших поставщиков услуг по государственному пенсионному планированию и страхованию домашних животных. На 2014 г. капитал составлял 195 миллиардов долларов, а доход — 24 миллиарда долларов. С 2005 г. компания начала вводить принципы Agile и бережливого производства, чтобы улучшить результаты работы 5000 своих сотрудников и позволить им вводить инновации по своей инициативе.

Стив Фарли, директор по информационным технологиям, вспоминает: «В то время начали появляться интересные конференции, например Национальная конференция по гибкой разработке

(Agile national conference). В 2011 г. руководство Nationwide согласилось, что мы должны создать свою конференцию, получившую название TechCon. Организовывая это мероприятие, мы хотели улучшить методики обучения, а также сделать так, чтобы все на этой конференции происходило в контексте нашей компании, вместо того чтобы посыпать сотрудников на другие конференции».

Банк Capital One, один из крупнейших в США, чей капитал в 2015 г. составлял 298 миллиардов долларов, а доход — 24 миллиарда долларов, поставил перед собой цель — создать технологическую организацию мирового уровня. В 2015 г. банк провел первую внутреннюю конференцию по разработке программного обеспечения. Ее задача заключалась в том, чтобы построить культуру сотрудничества, наладить связи между профессионалами внутри компании и ускорить обучение. У конференции было 13 учебных программ и 52 сессии, принимало участие более 1200 сотрудников.

Тапабрата Пал, сотрудник Capital One и один из организаторов мероприятия, пишет: «У нас даже был выставочный зал с 28 стендами, где команды Capital One показывали потрясающие вещи. Мы специально решили, что там не должно быть других поставщиков программного обеспечения, потому что мы хотели, чтобы фокус был на целях Capital One».

Организация Target — шестая по размеру компания розничной торговли США, в 2014 г. ее доход составлял 72 миллиарда долларов, в 1799 магазинах, разбросанных по всему миру, работало 347 сотрудников. Росс Клэнтон и Хизер Микман, директор по разработке Target, с 2014 г. организовали шесть внутренних мероприятий DevOpsDays, проведенных по образцу конференции DevOpsDays 2013 г. компании ING. В них приняло участие более 975 сотрудников технологического сообщества Target.

После того как Микман и Клэнтон посетили конференцию DevOps Enterprise Summit в 2014 г., они провели собственную внутреннюю конференцию, пригласив докладчиков из других фирм: те могли бы поделиться своим опытом с руководством фирмы. Клэнтон отмечает: «Именно в 2015 г. мы смогли обратить на себя внимание руководителей и раскрутить наше дело. После этого мероприятия к нам подходило много людей, спрашивавших, как они могут присоединиться к нам и чем могут помочь».

Создание внутренней структуры, занимающейся консультированием и наставничеством, — один из самых известных методов распространения знаний внутри организации. Эта идея может воплощаться разными способами. В Capital One у профильных специалистов есть свои приемные часы, когда любой желающий может проконсультироваться с ними, задать вопросы и так далее.

Ранее в этой книге мы рассказывали о том, как групплет тестирования (Testing Grouplet) создал в Google культуру автоматизированного тестирования мирового уровня. История на этом не закончилась: сотрудники продолжили совершенствовать систему автоматизированного тестирования с помощью таких инструментов, как блиц-обучения, наставничество и даже внутренняя программа сертификации.

По словам Бланда, в то время в Google была принята политика выделения 20 % времени на инновации. Разработчики могли тратить примерно один день в неделю на проекты Google, не связанные с их основным кругом задач. Некоторые инженеры объединились в групплеты — спонтанно сложившиеся группы единомышленников. Они хотели вместе использовать свои 20 % времени для концентрированных блиц-обучений.

Основателями первого тестового групплета были Бхарат Медиратта и Ник Лесецки. Цель — продвижение автоматизированного тестирования в Google. Хотя у них не было своего бюджета или формального начальства, но, как сказал Майк Бланд, «не было и никаких явных ограничений. И мы этим воспользовались».

Для внедрения команда пользовалась несколькими механизмами, но самым известным из них стало «тестирование в уборной» (Testing on the Toilet, TotT), их еженедельный журнал. Каждую неделю почти во всех уборных Google по всему миру появлялась почтовая рассылка. Бланд замечал: «Целью было распространение знаний о тестировании по всей компании. Вряд ли обычная онлайн-рассылка смогла бы заинтересовать кого-то до такой степени».

Бланд продолжает: «Один из самых значимых релизов TotT назывался Test Certified: Lousy Name, Great Results («Аттестация в тестировании: дурацкое название, отличные результаты»), в нем описывались два мероприятия, внесших огромный вклад в распространение автоматизированного тестирования».

В поэтапном плане Test Certified (TC) описывались шаги по улучшению автоматизированного тестирования. По словам Бланда, «Нашим намерением было использовать культуру Google, сфокусированную на наблюдении за разными показателями... и преодолеть первое страшное препятствие — непонимание того, откуда или как начать. На уровне 1 нужно было быстро определить основной показатель, на уровне 2 — сформулировать стратегию и выполнить план по покрытию тестами, целью уровня 3 было достижение долговременной цели по полноте охвата».

Второй важный шаг заключался в том, чтобы каждая команда могла воспользоваться помощью или советами наставника по программе ТС или тест-наемников (команды штатных коучей и консультантов компаний), работающих непосредственно с командами над улучшением качества кода и тестов. Для этого наемники применяли знания, инструменты и методики группплета тестирования к коду команд, используя ТС и как руководство, и как конечную цель. Сам Блэнд был руководителем этого группплета с 2006 по 2007 г. и одним из тест-наемников с 2007 по 2009 г.

Блэнд отмечает: «Нашей целью было привести все команды на уровень 3, участвовали ли они в нашей программе или нет. Мы также тесно сотрудничали с командой по разработке внутренних инструментов тестирования, давая им обратную связь по сложным ситуациям, знакомым по ситуациям в других командах. Мы были главной наступательной силой в этой борьбе, упорно продвигая наши инструменты, и в итоге фраза: "У нас нет времени на тестирование" перестала быть оправданием».

Далее он продолжает: «Уровни ТС использовали культуру метрической системы Google — три уровня тестирования, то, что люди могли обсуждать между собой и чем они могли хвастаться во время обзора эффективности работы. Группплет тестирования в итоге смог выбрать финансирование для тест-наемников, штатной команды внутренних консультантов. Это важный шаг, потому что руководство теперь полностью на нашей стороне, не только с указаниями, но и с настоящим финансированием».

Другим важным шагом было внедрение блиц-обучений «исправь это» (FixIt), охватывающих всю компанию. Блэнд описывает блиц-обучения так: «Когда обычные инженеры с идеей и чувством долга вербуют всех инженеров Google на однодневные интенсивные исправления кода и внедрение новых инструментов». Он организовывал четыре таких мероприятия, охватывавших всю компанию: два блица в тестировании и два блица, связанных с инструментами. В последнем приняло участие более 100 добровольцев из более чем 20 филиалов в 13 странах. Он также возглавлял группплет «Исправь это» с 2007 по 2008 г.

Узконаправленные блиц-обучения «исправь это», по словам Блэнда, должны проводиться в самые важные моменты, чтобы воодушевлять и давать энергию людям на решение важных проблем. С каждым значительным вложением сил масштабная цель изменить культуру компании становится все ближе и ближе.

Польза культуры тестирования очевидна: достаточно посмотреть на потрясающие результаты компании Google, многократно описанные в этой книге.

В этой главе рассказывалось о создании ритуалов, помогающих укреплять отношение к жизни как к непрерывной учебе и понимание того, что улучшения в ежедневной работе ценнее, чем сама работа. Мы добиваемся этого, выделяя время на важные дела, создавая форумы, где все желающие могут учиться сами и учить других, как внутри нашей организации, так и снаружи. Мы помогаем командам находить экспертов и учиться у них, с помощью коучинга и консультирования или просто с помощью специально выделенных приемных часов, когда сотрудники могут получить ответы на вопросы у знатоков своего дела.

Когда все помогают друг другу учиться в ходе повседневной работы, компания начинает быстро опережать конкурентов и в результате отвоевывает рынок. Но важнее все-таки то, что мы помогаем друг другу раскрыть свой истинный потенциал.

На протяжении части V мы изучили методики для создания в компании культуры обучения и экспериментирования. Обучение на ошибках, создание единых баз знаний и обмен опытом крайне важны, когда мы работаем в сложных системах; это делает нашу культуру более беспристрастной, а наши системы — более безопасными и устойчивыми.

В части VI мы рассмотрим, как расширить и усилить поток ценности, обратную связь, обучение и экспериментирование, используя их для достижения целей информационной безопасности.

## **Введение**

В предыдущих главах мы рассмотрели создание быстрого потока ценности от отправки готового кода в систему до релиза, а также противоположно направленного потока обратной связи. Мы изучили особенности культуры организации, ускоряющие обучение и распространение новых знаний и усиливающие слабые сигналы о сбоях, благодаря чему рабочая среда становится еще более безопасной.

В части VI мы продолжим рассматривать эти вопросы, при этом учитывая цели не только разработки и эксплуатации, но и отдела информационной безопасности. Это поможет нам повысить степень конфиденциальности, надежности и доступности наших сервисов и данных.

Вместо того чтобы вспоминать о безопасности только в самом конце проекта, мы будем создавать и интегрировать средства контроля защиты в ходе повседневной работы в разработке и эксплуатации, чтобы в итоге за безопасность отвечали все сотрудники компании. В идеале эта работа должна быть автоматизирована и встроена в конвейер развертывания. Кроме того, мы снабдим автоматическим управлением методики, предполагающие действия вручную, способы принятия решений и процессы согласования, чтобы меньше полагаться на разделение обязанностей и процессы одобрения изменений.

Автоматизируя эти действия, мы можем в любой момент продемонстрировать аудиторам, экспертам или коллегам, что средства контроля работают эффективно.

В итоге мы не только улучшим безопасность систем, но и создадим такие процессы, благодаря которым будет проще проводить аудит или оценивать эффективность контроля. Они будут соответствовать законодательным и контрактным требованиям. Чтобы достичь этой цели, мы:

- сделаем безопасность частью повседневной работы всех сотрудников;
- встроим профилактические меры безопасности в единый репозиторий исходного кода;
- преобразуем конвейер развертывания в соответствии с принципами безопасности;
- преобразуем телеметрию с учетом принципов безопасности, чтобы улучшить возможности обнаружения сбоев и восстановления после ошибок;
- обеспечим защиту конвейера развертывания;
- совместим процедуры развертывания и процессы согласования изменений;
- уменьшим зависимость от разделения обязанностей.

Делая заботу о безопасности частью повседневной работы и деля ответственность за нее между всеми сотрудниками, мы улучшаем безопасность организации. Правильная забота о защите данных означает, что мы ответственно и благородно обращаемся с конфиденциальной информацией. Это, в свою очередь, означает, что наша компания надежна и более доступна для клиентов, поскольку способна поддерживать непрерывность своей работы и легко восстанавливаться после неудач. Мы также можем устранять проблемы безопасности до того, как они приведут к катастрофическим результатам, а также повысить предсказуемость работы наших систем. И, наверное, самое важное — мы можем поднять защиту систем и данных на качественно новый, ранее недостижимый уровень.

## **Глава 22. Защита информации как часть повседневной работы всех сотрудников компании**

Одним из главных возражений против принятия принципов и практик DevOps стало то, что службы безопасности, как правило, не позволяют их использовать. И тем не менее подход DevOps является одним из лучших, если нужно встроить защиту информации в повседневную работу всех участников потока создания ценности.

Когда за информационную безопасность отвечает отдельное подразделение, возникает много проблем. Джеймс Уикетт, один из создателей средства защиты GauntIt и организатор конференции DevOpsDays в Остине и конференции Lonestar Application Security, отмечает:

«Одной из причин появления DevOps называют необходимость повысить производительность разработчиков, потому что с ростом числа разработчиков инженеры эксплуатации перестают справляться с развертываниями. Это ограничение еще сильнее бросается в глаза в информационной безопасности: отношение инженеров разработки, эксплуатации и информационной безопасности в типичной организации — обычно 100:10:1. Когда работников информационной безопасности так мало, без автоматизации и интегрирования защиты информации в повседневную деятельность отделов разработки и эксплуатации их времени будет хватать только на проверку, соответствует ли продукт нормативам и требованиям, что прямо противоположно принципам защиты данных. И, кроме того, из-за этого все нас ненавидят».

Джеймс Уикетт и Джош Кормен, бывший технический директор компании Sonatype и известный исследователь информационной безопасности, писали о встраивании целей защиты информации в DevOps, о наборе методик и принципов под названием *прочный DevOps* (Rugged DevOps). Похожие идеи, где защита информации встроена во все стадии цикла создания ПО, были разработаны Тапабратой Палом, директором и инженером по разработке платформ банка Capital One, и командой Capital One. Они назвали придуманные ими процессы DevOpsSec. Один из источников идей DevOps — книга *Visible Ops Security*, ее авторы — Джин Ким, Пол Лав и Джордж Спаффорд.

В этой книге мы изучали, как полностью встроить цели контроля качества и эксплуатации на протяжении всего потока создания ценности. В этой главе мы пишем, как похожим способом можно встроить цели информационной безопасности в повседневную работу, увеличив продуктивность разработчиков и инженеров эксплуатации, безопасность и надежность систем и сервисов.

Одна из наших целей — сделать так, чтобы инженеры службы безопасности начинали работать с командами разработчиков как можно раньше, а не присоединялись в самом конце проекта. Один из возможных способов добиться этого — приглашать службу безопасности на презентации продукта в конце каждого цикла разработки, чтобы ей было проще разобраться в целях разработчиков в контексте целей организации, пронаблюдать за развитием компонентов продукта, высказать пожелания и дать обратную связь на ранних стадиях проекта, когда еще достаточно времени и свободы, чтобы вносить правки.

Джастин Арбакл, бывший ведущий архитектор GE Capital, отмечает: «Когда дело дошло до информационной безопасности и соответствия нормам и стандартам, мы обнаружили, что проблемы в конце проекта обходились гораздо дороже, чем в начале, и проблемы безопасности были худшими. “Проверка соответствия требованиям на промежуточных этапах” стала одним из важных ритуалов равномерного распределения сложности по всем стадиям создания продукта».

Далее он продолжает: «Вовлекая службу безопасности в создание любого нового элемента функциональности, мы сильно сократили число чек-листов и начали больше полагаться на знания и опыт инженеров информационной безопасности на всех стадиях разработки продукта».

Это упростило выполнение бизнес-целей организации. Снехал Антани, бывший директор по информационным технологиям подразделения Enterprise Architecture компании GE Capital Americas, говорил, что у них тремя ключевыми показателями были «скорость разработки (то есть скорость вывода на рынок новых продуктов и элементов функциональности), неудачи в работе с клиентами (то есть сбои и ошибки) и время на выполнение запроса о соответствии нормам (то есть время от запроса аудиторов до предоставления всей количественной и качественной информации для удовлетворения запроса)».

Когда отдел безопасности вовлечен в процесс создания продукта, даже если его просто держат в курсе того, что происходит, сотрудники имеют представление о контексте работы и поэтому могут принимать взвешенные решения. Кроме того, инженеры безопасности могут помочь командам разработчиков понять, что нужно, чтобы продукт соответствовал нормам безопасности и законодательным требованиям.

Если есть возможность, стоит отслеживать все проблемы с защитой данных в той же системе учета, которая используется в разработке и эксплуатации. Такой подход очень отличается от

традиционного подхода службы безопасности, где все уязвимые места хранятся в инструменте GRC (governance, risk, compliance — управление, риск, соответствие требованиям), а доступ к нему есть только у отдела безопасности. Вместо этого будем помещать все задачи по защите данных в системы, используемые разработкой и эксплуатацией.

В презентации 2012 г. на конференции DevOpsDays в Остине Ник Галбрет, на протяжении многих лет возглавлявший отдел информационной безопасности в Etsy, так описывает свой подход к проблемам защиты: «Все задачи по защите данных мы создавали в JIRA. Ею все инженеры пользуются каждый день, и их приоритет был либо “P1”, либо “P2” — то есть они должны были быть исправлены сразу же или к концу недели, даже если проблема была во внутреннем приложении организации».

Кроме того, он утверждает: «Каждый раз, когда у нас была проблема с безопасностью, мы проводили совещание по разбору ошибок, потому что после них инженеры лучше понимали, как предотвратить такие ошибки в будущем, и потому что это отличный способ распространять знания о безопасности и защите данных между командами».

В мы создали хранилище кода, доступное всей организации. В нем все могут легко найти накопленные знания организаций: не только о коде, но и о наборах инструментов, о конвейере развертывания, стандартах и т. д. Благодаря этому все сотрудники могут извлечь пользу из опыта своих коллег.

Теперь добавим в репозиторий все механизмы и инструменты, которые помогут сделать приложения и среды более безопасными. Мы добавим одобренные службой безопасности библиотеки, выполняющие вполне конкретные цели по защите данных: это библиотеки и сервисы аутентификации и шифрования данных. Поскольку все в потоке создания ценности DevOps используют системы контроля версий для всего, что пишут или поддерживают, добавление компонентов и утилит безопасности позволит влиять на повседневную деятельность разработки и эксплуатации, потому что теперь все, что мы создаем, становится доступно для поиска и повторного использования. Система контроля версий также служит многонаправленным средством коммуникации, благодаря чему все находятся в курсе сделанных изменений.

Если сервисы в нашей организации централизованные и общедоступные, мы можем создавать и управлять общедоступными платформами и в сфере безопасности, например централизовать аутентификацию, авторизацию, логирование и другие нужные разработке и эксплуатации сервисы защиты и аудита. Когда инженеры используют одну из заранее созданных библиотек или сервисов, им не нужно планировать специальную проверку безопасности для этого модуля. Укрепление защиты конфигураций, настройки безопасности баз данных, длина ключей — везде будет достаточно просто воспользоваться уже созданными ориентирами.

Чтобы еще больше увеличить шансы, что библиотеки будут использоваться правильно, можно провести специальный тренинг для команд разработки и эксплуатации, а также отрецензировать их работу и проверить, что требования безопасности выполнены, особенно если команды пользуются этими инструментами впервые.

Наша итоговая цель — предоставить библиотеки и сервисы защиты информации, нужные в любом современном приложении или среде: это аутентификация, авторизация, управление паролями, шифрование данных и так далее. Кроме того, мы можем создать эффективные настройки конфигурации безопасности для компонентов, используемых командами разработки и эксплуатации в своих стеках приложения, например логирование, аутентификация и шифрование. Сюда можно включить вот что:

- библиотеки и рекомендуемые конфигурации (например, 2FA (двуухфакторная аутентификация), хэширование паролей bcrypt, логирование);
- управление секретной информацией (например, настройки соединения, ключи шифрования) с помощью таких инструментов, как Vault, sneaker, Keywhiz, credstash, Trousseau, Red October и так далее;
- пакеты и сборки операционных систем (например, NTP для синхронизации времени, безопасные версии OpenSSL с правильными конфигурациями, OSSEC или Tripwire для слежения за целостностью файлов, конфигурации syslog для логирования важных параметров безопасности в стек ELK).

Помещая все эти инструменты в единый репозиторий, мы упрощаем инженерам работу: они могут использовать в приложениях и средах гарантированно верные стандарты логирования и шифрования, без лишних усилий с нашей стороны.

Кроме того, можно скооперироваться с командами эксплуатации и создать базовый справочник и образы нашей операционной системы, баз данных и другой инфраструктуры (например, NGINX, Apache, Tomcat), чтобы проконтролировать, что они находятся в безопасном, безрисковом, известном состоянии. Единый репозиторий становится тем местом, где можно не только взять последние версии кода, но и где мы сотрудничаем с другими инженерами, наблюдаем за важными с точки зрения безопасности модулями и в случае непредвиденных происшествий получаем соответствующие оповещения.

В прошлые эпохи при работе над безопасностью приложения анализ его защиты обычно начинался после завершения разработки. Зачастую результат такого анализа — сотни страниц текста в формате PDF с перечислением уязвимых мест. Их мы отправляли командам разработки и эксплуатации. Эти проблемы обычно оставлялись без внимания, так как дедлайны подходили уже совсем близко или уязвимые места обнаруживались слишком поздно, чтобы их можно было легко исправить.

На этом шаге мы автоматизируем столько тестов защиты данных, сколько сможем, чтобы они проводились вместе со всеми другими тестами в конвейере развертывания. В идеале любое подтверждение кода и в разработке, и в эксплуатации должно приводить к запуску тестирования, даже на самых ранних стадиях проекта.

Наша цель — дать разработке и эксплуатации быструю обратную связь по их работе, чтобы они были в курсе, когда их код потенциально небезопасен. Благодаря этому они смогут быстро находить и исправлять проблемы с защитой данных в ходе ежедневной работы, накапливая полезный опыт и предотвращая будущие ошибки.

В идеале автоматизированные тесты должны запускаться в конвейере развертывания вместе с другими инструментами анализа кода.

Такие инструменты, как GauntIt, специально были созданы для работы в конвейере развертывания. Они проводят автоматические тесты безопасности для приложений, зависимостей приложений, для среды и так далее. Примечательно, что GauntIt даже помещает все свои тесты в тестовые скрипты на языке Gherkin, широко используемом разработчиками для модульного и функционального тестирования. Благодаря такому подходу тесты оказываются в уже знакомой им среде разработки. Кроме того, их можно легко запускать в конвейере развертывания при каждом подтверждении изменений кода, например при статическом анализе кода, проверке на наличие уязвимых зависимостей или при динамическом тестировании.

Jenkins					
S	W	Name	Last Success	Last Failure	Last Duration
●	●	Static analysis scan	7 days 1 hr - #2	N/A	6.3 sec
●	●	Check known vulnerabilities in dependencies	N/A	7 days 1 hr - #2	1.6 sec
●	●	Download and unit test	7 days 1 hr - #2	N/A	32 sec
●	●	Scan with OWASP ZAP	7 days 1 hr - #2	N/A	4 min 43 sec
●	●	Start	7 days 1 hr - #2	N/A	5 min 46 sec
●	●	Virus scanning	7 days 1 hr - #2	N/A	4.7 sec

Рис. 43. Автоматизированное тестирование в системе Jenkins (источник: Джеймс Уикет и Гарет Рашгров, “Battle-tested code without the battle,” презентация на конференции Velocity 2014, выложенная на сайте , 24 июня 2014 г., )

Благодаря этим методикам у всех в потоке ценности будет мгновенная обратная связь о статусе безопасности всех сервисов и продуктов, что позволит разработчикам и инженерам эксплуатации быстро находить и устранять проблемы.

В тестировании на стадии разработки часто обращают особое внимание на правильность работы приложения, концентрируясь на потоках «положительной логики». Такой тип тестирования часто называют *счастливым путем* (happy path), в нем проверяется обычное поведение пользователя (и иногда некоторые альтернативные пути), когда все происходит, как было запланировано, без исключений и ошибок.

С другой стороны, хорошие тестировщики, инженеры службы безопасности и специалисты по фродо-мешенничеству часто заостряют внимание на *грустных путях* (sad path), когда что-то идет не так, особенно если это касается ошибок, связанных с защитой данных (такие виды типичных для защиты данных событий часто в шутку называют *плохими путями*, или bad path).

Например, предположим, что на нашем сайте есть форма и в нее покупатель для оплаты заказа вносит данные своей платежной карты. Мы хотим найти все грустные и плохие пути, убедиться в том, что недействительные данные карты не принимаются, и предотвратить возможное использование уязвимых мест сайта, например внедрение SQL-кода, переполнение буфера и так далее.

Вместо того чтобы проводить проверку вручную, в идеале стоит сделать тесты безопасности частью автоматизированных тестовых модулей, чтобы они могли проводиться в конвейере развертывания непрерывно. Скорее всего, нам потребуются следующие виды функциональности.

- **Статический анализ:** это тестирование не в среде исполнения: идеальная среда — конвейер развертывания. Обычно инструмент статического анализа проверяет код программы на все возможные способы выполнения кода и ищет недочеты, лазейки и потенциально вредоносный код (иногда этот процесс называют «тестированием изнутри наружу»). Примеры инструментов статического анализа — Brakeman, Code Climate и поиск запрещенных функций (как, например, «exec()»).
- **Динамический анализ:** в противоположность статическому тестированию динамический анализ состоит из тестов, проводимых во время работы программы. Динамические тесты проверяют системную память, поведение функций, время ответа и общую работоспособность системы. Этот метод (иногда называемый «тестированием снаружи внутрь») имеет много общего с тем, как злоумышленники взаимодействуют с приложением. Примерами инструментов могут быть Arachni и OWASP ZAP (Zed Attack Proxy). Некоторые типы тестирования на проникновение (penetration testing) можно проводить автоматически, их можно включить в динамический анализ с помощью таких инструментов, как Nmap и Metasploit. В идеале автоматизированное динамическое тестирование должно проводиться во время фазы автоматического функционального тестирования в конвейере развертывания или даже уже во время эксплуатации сервиса. Чтобы проконтролировать соблюдение требований безопасности, можно настроить инструменты вроде OWASP ZAP, так чтобы они атаковали наши сервисы через прокси-сервер, а затем изучать сетевой трафик в специальной тестовой программе.
- **Проверка зависимостей:** еще один тип статического тестирования, проводимый во время сборки программы в конвейере развертывания, включает в себя проверку всех зависимостей продукта на наличие бинарных и исполняемых файлов, а также проверку того, что у всех этих зависимостей (а над ними у нас часто нет контроля) нет уязвимых мест или вредоносного кода. Примеры инструментов для таких тестов — Gemnasium и bundler audit для Ruby, Maven для Java, а также OWASP Dependency-Check.
- **Целостность исходного кода и подпись программы:** у всех разработчиков должен быть свой PGP-ключ, возможно, созданный и управляемый в такой системе, как keybase.io. Все подтверждения кода в системе контроля версий должны быть подписаны — это легко сделать с помощью таких инструментов свободного ПО, как gpg и git. Кроме того, все пакеты, созданные во время процесса непрерывной интеграции, должны быть подписаны, а их хэш должен быть записан в централизованный сервис логирования для облегчения аудита.

Кроме того, стоит определить шаблоны проектирования для написания такого кода, злоупотребить которым будет непросто. Это, например, введение ограничений для скорости сервисов и отключение кнопок отправления ранее использованных данных. Организация OWASP публикует много полезных руководств, в том числе серию чек-листов, рассматривающих следующие вопросы:

- как хранить пароли;
- как работать с забытыми паролями;
- как работать с входом в систему;
- как избежать уязвимых мест межсайтового скриптинга (XSS).

Презентация 10 Deploys per Day: Dev and Ops Cooperation at Flickr, подготовленная Джоном Оллспоу и Полом Хэммондом в 2009 г., известна тем, какое масштабное влияние она оказала на сообщество

DevOps. Ее эквивалент для сообщества информационной безопасности — по-видимому, презентация Джастина Коллинса, Алекса Смолина и Нила Мататала, представленная на конференции AppSecUSA в 2012 г. В ней рассказывалось о трансформации работы над защитой данных в компании Twitter.

Из-за стремительного роста у этой организации появилось много проблем. На протяжении многих лет, когда у сайта не было достаточного количества ресурсов, чтобы соответствовать высокому спросу пользователей, на экран выводилась страница с сообщением об ошибке. На ней был нарисован кит, поднимаемый в небо восемью птичками. Масштаб роста числа пользователей поражал воображение: между январем и мартом 2009 г. число активных пользователей Twitter выросло с 2,5 миллиона до 10 миллионов человек.

У компании в этот период также были и проблемы с безопасностью. В начале 2009 г. произошло два серьезных инцидента. В январе был взломан аккаунт . Затем, в апреле, с помощью словарного перебора были взломаны аккаунты администрации Twitter. Эти события привели к тому, что Федеральная комиссия по торговле США приняла решение о том, что организация вводила своих пользователей в заблуждение относительно безопасности их аккаунтов, и издала соответствующее распоряжение.

Согласно этому распоряжению, компания должна была в течение 60 дней ввести несколько мер, а затем поддерживать их 20 лет. Среди этих мер были:

- назначение сотрудников, ответственных за план информационной безопасности компании;
- определение относительно предсказуемых рисков, как внутренних, так и внешних, приводящих к несанкционированному проникновению, а также создание и воплощение плана по сокращению этих рисков;
- защита личной информации, причем не только от внешних источников, но и от внутренних, с примерным планом возможных источников верификации и тестирования безопасности и правильности этих реализаций.

Группа инженеров, назначенная для решения этой проблемы, должна была сделать защиту данных частью повседневной работы разработки и эксплуатации и закрыть дыры в системе безопасности — причины взломов аккаунтов.

В уже упоминавшейся презентации Коллинс, Смолин и Мататал определили несколько насущных проблем.

- **Предотвратить повторение ошибок в сфере безопасности:** они обнаружили, что на самом деле они снова и снова исправляли все те же дефекты и уязвимые места, что и раньше. Нужно было так изменить инструменты автоматизации и всю систему работы, чтобы эти проблемы перестали повторяться.
- **Встроить цели безопасности в уже существующие инструменты разработки:** они определили, что основным источником уязвимости были проблемы с кодом. Использовать какой-нибудь инструмент, выдающий огромный PDF-отчет, затем отправляющийся разработчикам или инженерам эксплуатации, было нерационально. Вместо этого требовалось средство, сообщающее конкретному разработчику, создавшему конкретную уязвимость, точную информацию, необходимую для устранения проблемы.
- **Сохранить доверие разработчиков:** им надо было заслужить и сохранить доверие разработчиков. Это означало, что требовалось узнавать о случаях, когда они посыпали разработчикам неверную информацию, чтобы стало возможным разобраться в причинах ложного срабатывания и впоследствии не тратить зря время разработчиков.
- **Поддерживать быстрый поток в информационной безопасности с помощью автоматизации:** даже когда сканирование кода на наличие уязвимых мест было автоматизировано, службе безопасности все равно приходилось делать много работы вручную и тратить время на ожидание. Нужно было ждать завершения сканирования, получить большую стопку отчетов, интерпретировать результаты и затем найти ответственного за исправление недочетов. А когда код менялся, нужно было проделывать все с самого начала. Автоматизируя работу вручную, сотрудники выполняли меньше примитивных задач, сводящихся к бездумному нажатию кнопок, и могли тратить время на более творческие задачи.

- **Отдать все, что связано с защитой данных на самостоятельное обслуживание, если это возможно:** они верили, что большинство специалистов хотят работать как можно лучше, поэтому нужно дать им доступ ко всему контексту и всей нужной информации, чтобы они могли сами исправлять ошибки.
- **Принять целостный подход к защите данных:** их целью был анализ со всех возможных точек зрения: исходный код, среда эксплуатации и даже то, что видели их клиенты.

Первый большой прорыв произошел во время общеорганизационного недельного хакатона, когда они интегрировали статический анализ кода в процесс сборки. Команда использовала Brakeman, сканирующий приложения Ruby on Rails на наличие уязвимых мест. Целью было встроить сканирование в самые ранние стадии разработки, а не только проверять итоговый код, отправляемый в репозиторий.

Результаты интеграции тестирования в процесс разработки были потрясающими. За год благодаря быстрой обратной связи о небезопасном коде и о способах его корректировки Brakeman сократил долю обнаруженных уязвимых мест на 60 %, как показано на рис. 44 (пики обычно соответствуют релизу новой версии Brakeman).

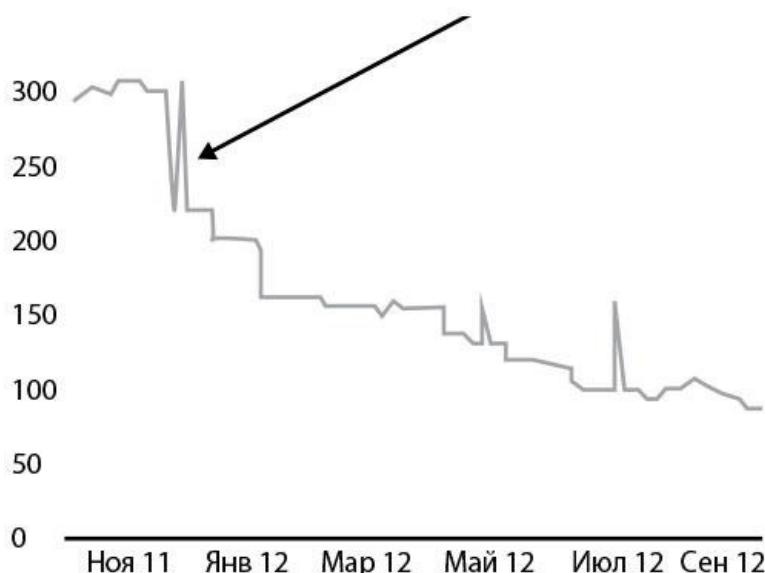


Рис. 44. Число уязвимых мест в системе безопасности, обнаруженнное Brakeman

Этот пример из практики показывает, как важно встроить защиту данных в повседневную работу и в инструменты DevOps и как эффективно это может работать. Благодаря такому подходу можно сократить риски безопасности, уменьшить вероятность появления уязвимых мест в системе и научить разработчиков писать более надежный код.

Джош Кормен отметил: как разработчики «мы больше не пишем программы для клиентов — вместо этого мы собираем их из программ с открытым исходным кодом, от которых стали очень зависимы». Другими словами, когда мы используем в продуктах какие-либо компоненты или библиотеки — как коммерческие, так и открытые, — то получаем не только их функциональность, но и их проблемы с безопасностью.

При выборе программного обеспечения мы определяем, когда наши проекты зависят от компонентов или библиотек, имеющих известные уязвимые места, и помогаем разработчикам делать этот выбор осознанно и тщательно, полагаясь на компоненты (например, на проекты с открытым исходным кодом), оперативно исправляющие бреши в защите данных. Мы также следим за разными версиями одной и той же библиотеки в сервисах, особенно за старыми, ведь в них есть уязвимые места.

Изучение нарушений конфиденциальности данных о владельцах карт наглядно показывает, насколько важна безопасность используемых компонентов. С 2008 г. ежегодный отчет Verizon PCI Data Breach Investigation Report (DBIR) считается самым авторитетным источником об утечках данных владельцев карт. В отчете 2014 г. они изучили более 85 000 утечек, чтобы лучше понять пути атак, как именно крадут данные владельцев карт и какие факторы способствуют нарушениям конфиденциальности.

Исследование DBIR показало, что в 2014 г. всего из-за десяти уязвимых мест (так называемые

распространенные уязвимые места и риски — CVE (common vulnerabilities and exposures)) произошло примерно 97 % утечек информации. Возраст восьми из этих уязвимых мест был больше десяти лет.

В отчете Sonatype State of the Software Supply Chain Report 2015 г. проанализированы данные об уязвимых местах репозитория Nexus Central Repository. В 2015 г. в этом репозитории хранилось более 605 000 проектов с открытым кодом, он обслуживал более 17 миллиардов запросов на скачивание программ и зависимостей, в основном для платформы Java, от более чем 106 000 организаций.

В отчете упоминались следующие пугающие факты:

- типичная организация полагалась на 7601 программный продукт (то есть источник программного обеспечения или компонент) и использовала 18 614 разных версий (то есть частей программного обеспечения);
- из этих компонентов у 7,5 % содержались известные уязвимые места, примерно 66 % были известны более двух лет, и никаких мер по их устраниению предпринято не было.

Последний факт подтверждается и в исследовании Дэна Джира и Джоша Кормэна: они показали, что из всех открытых проектов с известными уязвимыми местами, зарегистрированных в Национальной базе, проблемы с защитой были устранены только у 41 % и в среднем на выпуск патча требовалось 390 дней. Для уязвимых мест с наивысшей степенью опасности (уровень 10 по шкале CVSS) исправление заняло в среднем 224 дня.

На этом шаге мы должны сделать все возможное, чтобы наши среды были надежны и устойчивы, а связанные с ними риски — минимальны. Хотя мы уже могли создать понятные и хорошие конфигурации, нужно обязательно встроить средства мониторинга и контроля, чтобы все экземпляры в эксплуатации соответствовали этим конфигурациям.

Для этого мы создадим автоматизированные тесты, проверяющие, что у средств обеспечения устойчивости конфигураций, настроек безопасности баз данных, длин ключей и многое другое выставлены верные параметры. Кроме того, мы используем тесты для сканирования сред на известные уязвимые места.

Другое направление проверки безопасности — анализ того, как работают реальные среды (другими словами, «как есть», а не как «должно быть»). Примеры инструментов этого направления — Nmap, следящий за тем, что открыты только нужные порты, и Metasploit, проверяющий, что все типичные уязвимые места сред закрыты, например с помощью симуляции SQL-инъекций. Результаты применения этих инструментов должны храниться в общем репозитории и в процессе функционального тестирования сравниваться с предыдущими версиями. Это поможет быстро отследить любые нежелательные изменения.

В 2016 г. федеральные государственные учреждения США должны были потратить на информационные технологии 80 миллиардов долларов. Вне зависимости от учреждения, чтобы перевести систему в эксплуатацию, нужно было получить специальное разрешение на использование от соответствующего органа. Законы и требования в исполнительной власти состоят из десятков документов. В сумме они насчитывают более 4000 страниц, усеянных непонятными аббревиатурами вроде FISMA, FedRAMP и FITARA. Даже в системах с низким уровнем конфиденциальности, целостности и доступности должно быть проверено, задокументировано и проверено более сотни элементов. После завершения разработки обычно требуется от восьми до четырнадцати месяцев, чтобы получить разрешение.

Чтобы решить эту проблему, команда 18F Управления служб общего назначения федерального правительства приняла многоэтапный план. Майк Блэнд объясняет: «Группу 18F в Управлении служб общего назначения создали на волне ажиотажа от успеха восстановления сервиса ради полного изменения подхода правительства к разработке и покупке программного обеспечения».

Одним из результатов работы группы стала платформа, собранная из общедоступных компонентов с открытым исходным кодом. В настоящее время сервис работает в облаке AWS GovCloud. Платформа не только сама справляется с такими задачами эксплуатации, как логирование, мониторинг, оповещения и управление жизненным циклом сервиса. Ими обычно занимаются команды эксплуатации. Она также сама следит за большей частью задач по соблюдению законодательных требований. Для приложений, запускаемых на этом сервисе, большинство компонентов, обязательных для правительственные систем, можно реализовать на инфраструктурном или платформенном уровне. После этого документировать и тестировать нужно только компоненты на уровне приложения, что значительно сокращает нагрузку по соблюдению

требований и уменьшает сроки получения разрешений.

Сервис AWS одобрен для использования всеми правительственные системами всех типов, в том числе требующими высокого уровня конфиденциальности, целостности и доступности. К тому времени, когда вы будете читать эту книгу, скорее всего, сервис будет одобрен для всех систем, требующих среднего уровня конфиденциальности, целостности и доступности.

Кроме того, команда разрабатывает платформу для автоматического создания планов безопасности систем (system security plan, SSP), «полных описаний архитектуры системы, ее компонентов и общих средств обеспечения безопасности... обычно очень сложных и занимающих несколько сотен страниц». Они разработали прототип инструмента, названного Compliance Masonry, чтобы данные SSP можно было хранить в машиночитаемом формате YAML и автоматически преобразовывать в GitBooks- или PDF-документы.

Группа 18F исповедует принципы открытой работы и выкладывает всю свою работу в свободный доступ. Вы можете найти Compliance Masonry и компоненты в репозиториях GitHub команды 18F, вы даже можете запустить свой инстанс . Работа по созданию документации для SSP ведется в тесном сотрудничестве с сообществом OpenControl.

Маркус Сакс, один из исследователей утечек данных компании Verizon, отмечал: «На протяжении многих лет организации только месяцы спустя обнаруживали утечку данных о владельцах карт. Что еще хуже, утечка обнаруживалась не внутренними инструментами контроля, а чаще всего людьми, не работающими в компании. Обычно это был или бизнес-партнер, или клиент, заметивший мошеннические операции с картой. Одна из главных причин заключалась в том, что никто регулярно не просматривал логи».

Другими словами, внутренние средства контроля редко отслеживают нарушения безопасности вовремя или из-за слепых пятен в системе мониторинга, или потому, что никто в компании не изучает соответствующую телеметрию.

В мы обсуждали создание такой культуры: все участники потока создания ценности участвуют в распространении телеметрии, чтобы коллеги могли видеть результаты работы сервисов в эксплуатации. Кроме того, мы изучили необходимость обнаружения слабых сигналов о неполадках, чтобы можно было обнаруживать и исправлять проблемы до того, как они приведут к катастрофическим последствиям.

Сейчас же мы рассмотрим введение мониторинга, логирования и оповещений для защиты данных, а также проконтролируем их централизацию, чтобы полученную информацию было легко получать и анализировать.

Мы добьемся этого, встраивая телеметрию защиты данных в инструменты разработчиков, тестировщиков и инженеров эксплуатации, чтобы все в потоке создания ценности могли видеть, как сиды и приложения работают во враждебной среде, где злоумышленники все время пытаются воспользоваться уязвимыми местами систем, получить неавторизованный доступ, встроить лазейки, воспрепятствовать авторизованному доступу и так далее.

Распространяя сведения о том, как наши сервисы ведут себя при внешней атаке, мы напоминаем всем о необходимости иметь в виду риски безопасности и разрабатывать соответствующие контрмеры в ходе ежедневной работы.

Чтобы вовремя замечать подозрительное поведение пользователей, свидетельствующее о мошенничестве или неавторизованном доступе, мы должны создать в приложениях соответствующую телеметрию.

Примерами показателей могут быть:

- успешные и неуспешные входы в систему;
- смена паролей пользователей;
- изменение адреса электронной почты;
- изменения данных кредитной карты.

Например, ранним сигналом чьей-то попытки получить неавторизованный доступ с помощью перебора (брутфорса) может быть соотношение неудачных и удачных попыток входа в систему. И, конечно же, нужно создать соответствующие средства оповещения о важных событиях, чтобы

проблемы можно было быстро обнаружить и исправить.

Помимо телеметрии приложений, нам также нужно получать достаточно информации из сред, чтобы можно было отслеживать ранние сигналы о неавторизованном доступе, особенно в компонентах, работающих в неподконтрольной нам инфраструктуре (например, хостинги, облачная инфраструктура).

Мы должны следить и при необходимости получать оповещения о следующих событиях:

- изменения в операционной системе (например, в эксплуатации, в инфраструктуре разработки);
- изменение групп безопасности;
- изменения конфигураций (например, OSSEC, Puppet, Chef, Tripwire);
- изменения облачной инфраструктуры (например, VPC, группы безопасности, пользовательские привилегии);
- попытки XXS (cross-site scripting attacks, межсайтовый скриптинг);
- попытки SQLi (SQL-инъекций);
- ошибки серверов (например, ошибки 4XX и 5XX).

Также нужно проконтролировать, что у нас правильно настроено логирование и вся телеметрия отправляется в нужное место. Когда мы фиксируем атаки, то, помимо логирования самого факта атаки, можно также сохранять информацию о ее источнике и автоматически блокировать доступ, чтобы верно выбирать контрмеры.

В 2010 г. Ник Галбрет работал директором по разработке в компании Etsy, в его обязанности входили контроль над информационной безопасностью, предотвращение незаконных операций и обеспечение защиты личных данных. Галбрет определил мошенничество так: «Неверная работа системы, разрешающая недопустимый или непроверяемый ввод данных в систему, который приводит к финансовым потерям, краже или потере данных, сбоям в работе системы, вандализму или атакам на другую систему».

Для обеспечения защиты данных Галбрет не стал создавать отдельную систему контроля фрод-мошенничеств или новый отдел информационной безопасности. Вместо этого он встроил эти задачи в весь поток ценности DevOps.

Галбрет создал телеметрию защиты данных, отображающуюся вместе со всеми другими показателями разработки и эксплуатации. Ее любой инженер Etsy мог видеть каждый день.

- **Некорректное завершение программы в эксплуатации (например, ошибки сегментации, ошибки дампов памяти и так далее):** «Особенно интересно было, почему некоторые процессы, запускаемые с одного из IP-адресов, снова и снова приводили к падению всей нашей среды эксплуатации. Такими же интересными были и эти “ошибки 500: внутренняя ошибка сервера”. Это были сигналы о том, что кто-то использовал уязвимое место, чтобы получить неавторизованный доступ к нашим системам, и эту дыру надо срочно заделать».
- **Ошибки синтаксиса баз данных:** «Мы всегда искали ошибки синтаксиса баз данных в нашем коде — это были или потенциальные лазейки для SQL-инъекций, или разворачивающиеся прямо на наших глазах атаки. По этой причине ошибки синтаксиса мы исправляли незамедлительно, потому что это один из основных путей взлома систем».
- **Признаки SQL-инъекций:** «Это был на удивление простой тест: мы просто поставили оповещение о любом вводе команды UNION ALL в полях ввода пользовательской информации, потому что это практически всегда говорит о попытке взлома с помощью внедрения SQL-кода. Мы также добавили специальный тест, чтобы подобный тип неконтролируемого пользовательского ввода не принимался нашей системой».

На рис. 45 показан пример графика, доступного каждому разработчику компании. На нем отображается число потенциальных атак с помощью SQL-инъекций в среде эксплуатации. Как отмечает Галбрет, «ничто так не помогает разработчикам понять враждебность среды

эксплуатации, как возможность видеть атаки на их код в реальном времени».



Рис. 45. Разработчики Etsy могли видеть попытки SQL-инъекций в систему мониторинга Graphite (источник: "DevOpsSec: Applying DevOps Principles to Security, DevOpsDays Austin 2012", опубликовано Ником Галбретом, 12 апреля 2012 г., )

Он также добавляет: «Одним из результатов введения этого графика стало то, что разработчики осознали: атаки происходят все время! И это было потрясающе, потому что их образ мышления изменился, они стали думать о безопасности кода в процессе его написания».

Инфраструктура, поддерживающая непрерывную интеграцию и непрерывное развертывание, также становится возможной целью атаки. Например, если злоумышленник взламывает серверы, на которых работает конвейер развертывания, где содержатся параметры доступа к системе контроля версий, то он может украсть наш исходный код. Что еще хуже, если у конвейера есть доступ с правом записи, взломщик может внести вредоносные изменения в репозиторий контроля версий и, следовательно, внести вредоносные правки в приложения и сервисы.

Как отметил Джонатан Клаудиус, бывший старший тестировщик безопасности в организации TrustWave SpiderLabs, «серверы непрерывной сборки приложений и тестирования работают потрясающе, я сам их использую. Но я начал задумываться о том, как использовать CI и CD для внедрения вредоносного кода. Что привело меня к вопросу: где хорошее место для того, чтобы спрятать вредоносный код? Ответ очевиден: в юнит-тестах. На них никто не смотрит, и они запускаются каждый раз, когда кто-то отправляет свой код в репозиторий».

Это наглядно демонстрирует, что для адекватной защиты целостности приложений и сред нужно также позаботиться о защите конвейера развертывания. Среди возможных рисков — разработчики могут написать код, разрешающий неавторизованный доступ (с этим можно справиться с помощью тестирования и рецензирования кода, а также с помощью тестов на проникновение), или неавторизованные пользователи могут получить доступ к нашему коду или среде (это исправляется проверкой конфигураций на соответствие известным правильным образцам и эффективным созданием патчей).

Кроме того, чтобы защитить конвейер непрерывной сборки, интеграции или развертывания, можно ввести такие стратегии:

- защита серверов непрерывной сборки и интеграции и создание механизмов для их автоматического восстановления, чтобы их нельзя было взломать; все точно так же, как и для инфраструктуры, поддерживающей ориентированные на клиентов сервисы;
- анализ и оценка всех изменений в системе контроля версий, или с помощью парного программирования во время подтверждения кода, или с помощью рецензирования кода между подтверждением кода и добавлением его в основную ветку. Так в серверы непрерывной интеграции не будет поступать неконтролируемый код (например, тесты могут содержать вредоносный код, делающий возможным неавторизованный доступ);
- оснащение репозиториев инструментами для обнаружения подозрительных API-вызовов в коде тестов (например, тесты, требующие доступа к файловой системе или сетевым соединениям), добавляемых в репозиторий, возможно, с карантином подозрительного кода и его немедленным анализом;
- предоставление для каждого процесса непрерывной интеграции своего изолированного контейнера или виртуальной машины;
- проверка того, что параметры доступа контроля версий, используемые системой непрерывной интеграции, разрешают только чтение.

В этой главе мы описали пути интегрирования целей защиты данных во все этапы ежедневной работы сотрудников. Для этого нужно встроить компоненты информационной безопасности в уже созданные механизмы и проконтролировать, что все запрашиваемые среды достаточно укреплены, а связанные с ними риски — минимизированы. Эти цели можно выполнить с помощью встраивания тестирования защиты данных в конвейер развертывания и создания соответствующей телеметрии в тестовых и производственных средах. Благодаря этим мерам увеличиваются как продуктивность разработчиков и инженеров эксплуатации, так и общий уровень безопасности. На следующем шаге мы рассмотрим, какими способами можно защитить наш конвейер развертывания.

## Глава 23. Безопасность конвейера развертывания

В этой главе мы изучим, как можно защитить конвейер развертывания, а также достичь цели по защите данных и выполнению требований в нашей среде, включая управление изменениями и разделение обязанностей.

Практически в любой достаточно большой ИТ-организации есть свои процессы управления изменениями. Это главные средства уменьшить операционные риски и риски, связанные с безопасностью. В вопросах защиты данных менеджер по регуляторным вопросам и менеджеры по безопасности полагаются на процессы управления изменениями, и им обычно нужны доказательства того, что все изменения были должным образом одобрены.

Если наш конвейер развертывания организован правильно и риски развертывания низкие, большую часть правок не придется одобрять вручную, поскольку о безопасности заботятся такие средства контроля, как автоматизированные тесты и проактивный мониторинг процесса эксплуатации.

На этом шаге мы сделаем все необходимое, чтобы успешно встроить цели безопасности и соответствия требованиям во все существующие процессы управления изменениями. Эффективные методики контроля перемен учитывают, что с разными типами изменений связаны разные риски и что эти перемены должны контролироваться по-разному. Эти процессы определены в стандарте ITIL, разбивающем все изменения на три категории.

- **Стандартные изменения:** это изменения с низким риском, проводящиеся в соответствии с четким, проверенным процессом. Их можно одобрить заранее. Это ежемесячные обновления таблицы налоговых ставок или кодов стран, правки оформления и содержания сайтов, некоторые виды патчей приложений или операционных систем. Их работа легко предсказуема. Человеку, предлагающему такие изменения, не нужно одобрения для внедрения, этот процесс может быть полностью автоматическим и, следовательно, логируемым, так как все его детали легко отследить.

- **Нормальные изменения:** риски этих перемен выше, им требуется рецензирование или одобрение от выбранных специалистов. Во многих организациях эта ответственность ошибочно возлагается на комитет по изменениям (change advisory board, CAB) или на комитет по срочным изменениям (emergency change advisory board, ECAB). Однако у них может не быть достаточной компетенции, чтобы понять все последствия предлагаемых перемен, что часто ведет к недопустимо долгим срокам. Эта проблема особенно важна для крупных развертываний кода, содержащих сотни тысяч (и даже миллионы) строк нового кода, написанных сотнями разработчиков на протяжении нескольких месяцев. Для одобрения нормальных изменений у комитета, как правило, есть форма запроса для внесения изменений (request for change, RFC), и в ней определяется информация, нужная для принятия или отклонения этого запроса. В форме RFC обычно описываются предполагаемые результаты для бизнеса, планируемые функциональность и гарантии, анализ рисков и альтернатив, а также план воплощения в жизнь.

- **Срочные изменения:** высокорисковые изменения, их нужно провести как можно быстрее (например, выпуск важного патча безопасности, восстановление сервиса). Для этих изменений часто нужно одобрение высшего руководства, но всю документацию можно оформить постфактум. Ключевая цель методик DevOps — так улучшить обычный процесс внесения изменений, чтобы его можно было использовать и для срочных изменений.

В идеальной ситуации, когда конвейер развертывания надежен, у нас уже есть большой опыт быстрых, надежных и спокойных развертываний. На этом этапе мы должны договориться с инженерами эксплуатации и теми, кто отвечает за одобрение изменений: все перемены до этого момента были достаточно безрисковыми, чтобы определить их в категорию стандартных изменений. Это позволит нам проводить развертывания без лишних процедур по согласованию, хотя все изменения по-прежнему будут надлежащим образом фиксироваться.

Один из способов аргументировать утверждение, что у наших правок низкий риск, — продемонстрировать историю изменений за продолжительный временной отрезок (месяцы или кварталы) и полный список всех проблем эксплуатации за этот же период. Если доля успешных изменений будет высокой, а индекс MTTR — низким, можно уверенно утверждать, что наши средства контроля эффективно предотвращают ошибки развертывания и что мы можем быстро обнаруживать и исправлять возникающие проблемы.

Даже если наши изменения классифицированы как стандартные, их все равно нужно фиксировать в соответствующих системах управления изменениями (например, Remedy или ServiceNow). В идеале развертывания будут проводиться автоматически, с помощью инструментов конвейера развертывания и управления конфигурациями (например, Puppet, Chef, Jenkins), а результаты тоже будут записываться автоматически. Благодаря этому все сотрудники компании будут иметь доступ

к информации об изменениях.

Мы можем привязать эти записи к конкретным задачам в инструментах планирования работы (например, JIRA, Rally, LeanKit, ThoughtWorks Mingle), что позволит создать более широкий контекст изменений, к примеру соотнести их с дефектами компонентов функциональности, сбоями в эксплуатации или требованиями заказчиков. Этого можно легко добиться с помощью включения номеров тикетов из инструментов планирования в комментарии о регистрации кода в системе контроля версий. Благодаря такому подходу мы сможем связать конкретное развертывание с изменениями в системе контроля версий, а они, в свою очередь, связаны с тикетами инструментов планирования.

Создать такую систему отслеживания связей и контекста легко, много времени и сил у инженеров это не отнимет. Отсылок на пожелания заказчиков, формальные требования или дефекты должно быть достаточно, более подробные детали, такие как тикеты для каждого подтверждения кода в систему контроля версий, вряд ли будут полезными, они только создадут лишнюю нагрузку на повседневную работу сотрудников.

Правки, не классифицируемые как стандартные, будут считаться *нормальными изменениями*, то есть перед развертыванием должны быть одобрены хотя бы частью комитета по изменениям. В этом случае нашей целью тоже будет ускорение развертываний, даже если они не будут полностью автоматизированными.

Мы должны убедиться, что поданные запросы на изменения — наиболее полные и точные, чтобы у комитета была вся нужная информация для верной оценки. В конце концов, если наша заявка будет неясной, ее вернут нам на доработку, что только увеличит время на введение изменений и вызовет сомнения в том, что мы понимаем цели процесса управления изменениями.

Практически всегда можно автоматизировать создание полных и точных запросов, дополняя тикеты деталями того, что именно нужно исправить. Например, можно автоматически создавать тикет изменений в ServiceNow со ссылкой на требования заказчика в системе JIRA вместе со сборкой манифеста, результатами тестов конвейера развертывания и ссылками на исполняемые скрипты Puppet/Chef.

Поскольку наши предложения будут оцениваться вручную, очень важно описать контекст изменений. Сюда нужно включить причины правок (снабженные ссылками на элементы функциональности, дефекты или отчеты о неполадках), на кого эти изменения повлияют и что именно будет изменено.

Наша цель — представить доказательства, что после изменений системы будут работать так, как и предполагалось. Хотя текстовые поля формы запроса на изменения обычно предполагают свободную форму заполнения, нужно добавить в нее ссылки на машиночитаемые данные, чтобы другим было проще пользоваться и обрабатывать наши данные (это, например, ссылки на файлы JSON).

Во многих пакетах инструментальных средств это можно сделать автоматически. Например, инструменты Mingle и Go компании ThoughtWorks могут автоматически связывать такие данные, как список исправленных дефектов или завершенные элементы функциональности, вместе и добавлять их в запрос на изменения.

После подачи заявления члены комитета рассмотрят, проанализируют и вынесут вердикт относительно предложений. Если все пойдет хорошо, начальство оценит подробность и качество оформления заявки, потому что подтвердить достоверность нашей информации будет легко (например, с помощью ссылок из инструментов конвейера развертывания). В итоге целью должно быть богатое портфолио успешных изменений, чтобы впоследствии начальство согласилось классифицировать наши автоматизированные правки как безопасные стандартные изменения.

Компания была основана в 2000 г., ее целью было сделать управление взаимоотношениями с клиентами легкодоступным и легко поставляемым сервисом. Предложения организации были широко востребованы на рынке, результатом чего стало успешное IPO в 2004 г. К 2007 г. у компании было более 59 000 клиентов, в день ее сервисы обрабатывали сотни миллионов транзакций, а годовой доход достиг 497 миллионов долларов.

Однако в то же самое время их способность к разработке и выпуску новой функциональности упала практически до нуля. В 2006 г. у них было четыре больших релиза, но в 2007-м компания сделала всего лишь один релиз, несмотря на то что к тому времени число ее инженеров увеличилось. В результате число новых компонентов функциональности на команду падало, а периоды между большими релизами увеличивались.

И поскольку размер каждого релиза становился все больше, результаты развертываний все ухудшались. Картик Раджан, тогда работавший директором по организации инфраструктуры, в

своей презентации отмечал, что 2007 г. стал «последним, когда программное обеспечение создавалось и поставлялось с помощью каскадной методологии разработки, и именно тогда мы перешли на инкрементальный процесс поставки».

На конференции DevOps Enterprise Summit в 2014 г. Дэйв Мангот и Рина Мэтью описали занявшую много лет DevOps-трансформацию компании, начавшуюся в 2009 г. Согласно Манготу и Мэтью, применяя принципы и методики DevOps, в 2013 г. организация сократила среднее время развертывания с шести дней до пяти минут. В результате стало легче масштабировать ресурсы, что позволило их сервисам обрабатывать более миллиарда транзакций в день.

Одной из главных тем трансформации компании Salesforce было стремление сделать качество ответственностью всех сотрудников, вне зависимости от того, работали ли они в разработке, эксплуатации или информационной безопасности. Для этого встроили автоматизированное тестирование во все стадии создания приложений и сред, а также во все процессы непрерывной интеграции и развертывания и создали инструмент с открытым кодом под названием Rouster, чтобы проводить функциональное тестирование своих модулей Puppet.

Они также начали проводить *разрушающие тестирования* — этот термин используется в промышленном производстве для обозначения длительных испытаний продукта в самых суровых условиях, пока тестируемый предмет не сломается. Команда Salesforce начала регулярно тестировать свои сервисы под большой нагрузкой, пока они не выходили из строя. Это помогло понять причины и ход отказа систем и внести нужные корректизы. Неудивительно, что в результате качество работы сервисов в нормальных условиях стало гораздо лучше.

Служба информационной безопасности также работала вместе со службой обеспечения качества на ранних стадиях проекта, принимая участие в таких критических фазах, как разработка архитектуры и тестов, а также встраивая инструменты защиты данных в процесс автоматического тестирования.

Для Мангот и Мэтью одним из важнейших результатов этого сурового процесса было то, что группа по управлению изменениями сказала им, что «изменения инфраструктуры, сделанные с помощью Puppet, теперь будут классифицироваться как “стандартные изменения”, для их внедрения не нужно будет одобрения комитета». Кроме того, было отмечено, что «изменения, вносимые в инфраструктуру вручную, все равно должны будут проходить процедуру одобрения».

Благодаря проделанной работе они смогли не только интегрировать процессы DevOps и процессы управления изменениями, но также создали мотивацию для дальнейшей автоматизации внесения правок в инфраструктуру.

Десятилетиями мы использовали разделение обязанностей как одно из главных средств сокращения рисков ошибок в процессах разработки ПО. В большинстве моделей жизненных циклов разработки систем считалось обычной практикой давать предложенные разработчиками изменения на анализ программисту, отвечающему за ту или иную библиотеку, чтобы он составил рецензию и одобрил правки, прежде чем инженеры ввели бы эти правки в эксплуатацию.

Есть много других менее спорных примеров разделения обязанностей в работе эксплуатации, например, администраторы серверов могут смотреть логи, но не могут удалять их или редактировать, чтобы никто с правами специального доступа не мог удалить свидетельства мошенничества или других проблем.

Когда мы проводили развертывания относительно редко (например, ежегодно) и когда наша работа была менее сложной, разделение работы и передачи управления проектами были приемлемыми способами ведения бизнеса. Однако с ростом сложности систем и увеличением частоты развертываний появляется необходимость того, чтобы все сотрудники в потоке создания ценности могли быстро увидеть результаты своей работы.

Разделение обязанностей часто мешает этому, замедляя и сокращая обратную связь, полученную инженерами. Из-за этого они не могут брать на себя полную ответственность за качество работы, а способность компаний получать и накапливать знания значительно сокращается.

Принимая во внимание все вышесказанное, мы должны избегать разделения обязанностей как средства контроля везде, где это возможно. Вместо этого нужно выбрать такие средства контроля, как парное программирование, непрерывные проверки внесенного кода и рецензирование кода. Эти методы помогут нам следить за качеством нашей работы. Кроме того, если от разделения обязанностей избавиться все-таки нельзя, внедрение этих методов покажет, что с их помощью можно добиться тех же результатов.

Билл Месси работает руководителем разработки в компании Etsy и отвечает за приложение оплаты под названием ICHT (аббревиатура от I Can Haz Tokens). ICHT проводит заказы клиентов через набор внутренних приложений обработки платежей: они принимают онлайн-заказ, выделяют

информацию о платежной карте клиента, маркируют ее, отсылают платежной системе и завершают транзакцию.

Поскольку в предметную область среди CDE входят «люди, процессы и технологии, хранящие, обрабатывающие и передающие данные владельцев платежных карт или конфиденциальную аутентификационную информацию», в том числе любые связанные с ними системные компоненты, приложение ICHT также попадает в область регулирования PCI DSS.

Чтобы поддерживать стандарты PCI DSS, приложение ICHT физически и логически отделено от остальных сервисов Etsy и управляется отдельной командой разработчиков, инженеров баз данных, специалистов по сетям и инженеров эксплуатации. У каждого члена команды есть два ноутбука: один для ICHT (который настроен по-особому, чтобы соответствовать требованиям DSS; в нерабочее время он хранится в сейфе) и один для прочей работы.

Благодаря такому подходу мы смогли отделить среду CDE от остальной части Etsy, тем самым сильно ограничив ту область, где должны соблюдаться стандарты PCI DSS. Системы, составляющие CDE, отделены (и управляются) от других сред компании на физическом, сетевом, логическом уровнях и на уровне исходного кода. Кроме того, среда CDE управляется и сопровождается многофункциональной командой, отвечающей только за нее.

Чтобы соблюсти требования по соответствуанию кода, группе ICHT пришлось поменять свои методики непрерывной поставки. Согласно разделу 6.3.2 PCI DSS v3.1, команды должны анализировать весь специально разработанный код до передачи в эксплуатацию или пользование, чтобы идентифицировать следующие потенциальные уязвимые места (с помощью процессов, проводимых вручную или автоматизированных).

- Анализируются ли изменения кода кем-то другим, помимо самого автора кода, и владеет ли эксперт методиками анализа кода и его написания?
- Проверяется ли код на соответствие требованиям написания безопасного кода?
- Исправляются ли проблемные места кода до релиза?
- Проверяются и одобряются ли результаты анализа кода соответствующими специалистами до релиза?

Чтобы выполнить эти требования, команда поначалу решила назначить Мэсси ответственным за проверку изменений и за их развертывание в эксплуатацию. Разворты для анализа отмечались в JIRA, затем Мэсси просматривал их и выносил вердикт и, наконец, вручную отправлял их в эксплуатацию.

Это позволило Etsy выполнить требования PCI DSS и получить от оценщиков подписанный Отчет о соответствии требованиям. Однако в команде возникли серьезные проблемы.

Мэсси отмечает один проблематичный побочный эффект — это «уровень “изоляции” в команде ICHT. Такого нет ни в одной другой команде Etsy. С тех пор как мы ввели разделение обязанностей и другие средства контроля в соответствии с PCI DSS, в этой среде больше никто не мог быть инженером широкого профиля».

В результате, пока другие команды Etsy работали рука об руку и проводили развертывания уверенно и без проблем, Мэсси с грустью констатировал: «В среде PCI царили страх и нежелание проводить развертывания и поддерживать код, потому что никто не знал, что происходит за пределами его области стека приложений. Небольшие изменения в организации работы привели к созданию непробиваемой стены между разработчиками и инженерами эксплуатации и небывалой с 2008 г. напряженности. Даже если вы полностью уверены в своей области, невозможно быть уверенными в том, что чьи-то правки не сломают вашу часть стека».

Этот пример показывает: соответствие требованиям можно поддерживать в компании, придерживающейся принципов DevOps. Однако мораль этой истории в том, что все достоинства, связанные в нашем сознании с высокопроизводительными командами DevOps, на самом деле очень хрупки: даже если у команды богатая история сотрудничества, высокое доверие друг к другу и общие цели, она может столкнуться с проблемами, когда вводятся механизмы контроля, основанные на недоверии.

По мере того как организации постепенно вводят методики DevOps, напряженность между ИТ-индустрией и аудитом нарастает. Новые подходы DevOps бросают вызов традиционному пониманию аудита, контроля и сокращения рисков.

Как отмечает Билл Шинн, ведущий архитектор по обеспечению безопасности Amazon Web Services, «Суть DevOps — в наведении мостов между разработкой и эксплуатацией. В какой-то мере проблема пропасти между DevOps и аудиторами еще больше. Например, сколько аудиторов могут читать код и сколько разработчиков читало NIST 800-37 или закон Грэмма — Лича — Блайли? Это создает большой разрыв в знаниях, и сообщество DevOps должно помочь преодолеть этот разрыв».

Среди обязанностей Билла Шинна, ведущего архитектора по обеспечению безопасности Amazon Web Services, — демонстрация крупным корпоративным клиентам того, что их работа может соответствовать многочисленным законам и требованиям. За долгие годы количество компаний, с которыми ему довелось поработать, перевалило за тысячу, среди них — Hearst Media, GE, Phillips и Pacific Life, открыто заявлявшие, что пользуются общедоступными облаками в высокорегулируемых средах.

Шинн отмечает: «Одна из проблем была в том, что аудиторы привыкли работать методами, не очень хорошо подходящими для шаблонов DevOps. Например, если аудитор видел среду с десятком тысяч серверов, он традиционно просил сделать выборку из тысячи серверов, вместе со скриншотами материалов по управлению активами, настройками контроля доступа, данных по установке агентов, логов серверов и так далее.

Для физических сред это нормально, — продолжает Шинн. — Но когда инфраструктура — это код, а из-за автомасштабирования серверы все время то появляются, то исчезают, как можно сделать выборку? Те же самые проблемы и с конвейером развертывания, он очень отличается от традиционного процесса разработки программного обеспечения, где одна группа пишет код, а другая вводит его в эксплуатацию».

Далее он объясняет: «В работе аудиторов самым распространенным методом сбора информации все еще остаются скриншоты и CSV-файлы, заполненные параметрами конфигураций и логами. Наша цель — создать альтернативные способы представления данных. Они четко показывают аудиторам, что наши средства контроля удобны и эффективны».

Чтобы сократить этот разрыв, у него есть команды, вместе с аудиторами разрабатывающие средства контроля. Они пользуются итеративным подходом, используя одно средство контроля за один подход, чтобы определить, что нужно для аудиторских доказательств. Благодаря этому аудиторы могут по запросу получать всю нужную им информацию, когда сервис находится в эксплуатации.

Шинн утверждает, что лучший способ добиться этого — «послать все данные в системы телеметрии, такие как Splunk или Kibana. Так аудиторы смогут получить все, что им нужно, без посторонней помощи. Им не требуется запрашивать выборку из данных, вместо этого они заходят в Kibana и ищут нужные аудиторские доказательства за конкретный временной период. В идеале они очень быстро найдут свидетельства того, что наши средства контроля действительно работают».

Шинн продолжает: «Благодаря современному аудиторскому логированию, чатам и конвейерам развертываний мы добились небывалой прозрачности и видимости того, что происходит в производственной среде, особенно если сравнивать с тем, как раньше обстояли дела в эксплуатации. Вероятность ошибок и брешей в безопасности стала гораздо меньше. Главная задача теперь состоит в том, чтобы преобразовать все эти доказательства в то, что аудитор сможет понять».

Для этого нужно, чтобы технические требования формировались на основе реальных нормативных требований. Шинн объясняет: «Чтобы найти, что требует HIPAA с точки зрения обеспечения безопасности, вам нужно посмотреть раздел 160 части 45 Свода федеральных нормативных актов, проверить подразделы А и С раздела 164. Но и там вы не сразу найдете то, что ищете, вам придется читать до части “технические меры предосторожности и аудиторские средства контроля”. Только тогда вы увидите, что нужно определить отслеживаемые действия, связанные с информацией о пациентах, затем спроектировать и реализовать средства контроля, выбрать инструменты и только потом собрать и проанализировать нужные данные».

Шинн продолжает: «Как именно выполнить это требование — предмет обсуждения между специалистами по надзору за соблюдением требований, службой защиты данных и командами DevOps. Особенного внимания требуют вопросы предотвращения, обнаружения и исправления ошибок. Иногда эти проблемы можно решить с помощью параметров конфигурации системы контроля версий. А иногда это проблема контроля мониторинга».

Шинн приводит пример: «Можно воплотить одно из этих средств контроля с помощью AWS CloudWatch и затем протестировать, что это средство запускается с помощью одной строки. Кроме того, нужно показать, куда отправляются логи: в идеале мы складываем их в общую систему логирования, где можем связать аудиторские доказательства с актуальными требованиями контроля».

Способ решения этой проблемы представлен в документе DevOps Audit Defence Toolkit: в нем

описывается весь процесс аудита в вымышленной организации (Parts Unlimited из книги The Phoenix Project) от начала и до конца. Начинается он с рассказа о целях организации, ее бизнес-процессах, главных рисках и заканчивается описанием итоговых средств контроля и того, как руководство компании смогло успешно доказать, что эти средства контроля существуют и эффективно работают. В тексте также приводится список типичных возражений со стороны аудиторов и то, как с ними работать.

В документе описано, как можно разработать средства контроля для конвейера развертывания, чтобы сократить известные риски, и приведены примеры аттестации качества средств и рабочих продуктов контроля, демонстрирующих их эффективность. Описание намеренно было сделано как можно более общим по отношению ко всем целям управления контролем, включающим в себя поддержку точной финансовой отчетности, соблюдение нормативных требований (например, SEC SOX-404, HIPAA, FedRAMP, типовые договоры Европейского союза и нормативные положения SEC Reg-SCI), контрактные обязательства (например, PCI DSS, DOD DISA) и эффективный и действенный процесс эксплуатации.

Мэри Смит (имя вымышлено) возглавляет инициативную группу DevOps в подразделении крупной американской финансовой организации, занимающейся банковским обслуживанием физических лиц. Она заметила, что служба информационной безопасности, аудиторы и регулирующие органы часто полагаются только на анализ кода, чтобы обнаруживать факты мошенничества. Вместо этого, чтобы сократить риски, связанные с ошибками и защитой данных, им следовало бы больше полагаться на средства мониторинга работы сервисов в эксплуатации вкупе с автоматизированным тестированием, анализом кода и оценкой качества.

Смит отмечает:

«Много лет назад у нас был разработчик, оставивший лазейку в коде, развертываемом в наши банкоматы. В нужный момент он мог переводить банкоматы в профилактический режим и получать наличные деньги. Эту мошенническую схему мы раскрыли быстро, и не с помощью анализа кода. Такие способы обхода защиты сложно, даже практически невозможно обнаружить, когда у злоумышленника достаточно мотивации, навыков и благоприятных возможностей.

Однако мы смогли быстро обнаружить мошенничество во время регулярных встреч группы эксплуатации для анализа, когда кто-то заметил, что банкоматы в городе переходят в режим обслуживания не по расписанию. Мы раскрыли схему даже до запланированной проверки наличности в банкоматах, когда аудиторы сверяют реальное количество денег с проведенными транзакциями».

В этом примере из практики противозаконная операция произошла, несмотря на процессы управления изменениями и разделение обязанностей между разработкой и эксплуатацией, но обнаружение и исправление бреши безопасности стали возможными благодаря эффективной эксплуатационной телеметрии.

В этой главе мы обсудили методики, помогающие сделать заботу об информационной безопасности работой всех сотрудников компании. В этих методиках все цели по защите данных встроены в повседневную работу всех участников потока ценности. Благодаря такому подходу мы значительно улучшаем эффективность средств контроля, чтобы успешно предотвращать появление брешей в системе безопасности, а также быстрее их обнаруживать и устранять. Кроме того, мы можем сильно сокращать объемы работы, связанные с подготовкой и прохождением аудиторских проверок.

В главах этой части мы изучили то, как применять принципы DevOps в информационной безопасности и сделать заботу о защите данных частью ежедневной работы. Надежная система безопасности гарантирует, что мы разумно и осторожно обращаемся с данными, можем быстро оправиться от проблем, связанных с нарушением защиты информации, до того, как последствия станут катастрофическими, и — это важнее всего — можем поднять безопасность наших систем и данных на новую, казавшуюся невозможной высоту.

## **Призыв к действию. Заключение**

Мы завершили подробное изучение принципов и методик DevOps. Сегодня все технические руководители сталкиваются с проблемами безопасности, надежности и гибкости, происходят масштабные технологические изменения, постоянно случаются утечки данных и нужно срочно выводить на рынок новые продукты. DevOps предлагает решение всех этих проблем. Мы надеемся, что книга помогла вам составить подробное представление о поисках решения.

Мы говорили, что, если не прикладывать специальные усилия, между разработкой и эксплуатацией возникнет неизбежный конфликт, порождающий всё более серьезные проблемы, а они приведут к увеличению сроков создания новых продуктов, снижению качества, росту числа сбоев и неполадок, чрезмерному вниманию к срочным делам в ущерб важным, сокращению производительности труда и росту профессионального выгорания сотрудников.

Принципы и методики DevOps позволяют разрешить этот хронический конфликт. Мы надеемся, что вы увидите, как трансформация DevOps помогает создавать динамичные, ориентированные на обучение компании и быстрый поток, выводя стандарты надежности и безопасности на мировой уровень, а также усиливая конкурентоспособность и увеличивая удовлетворенность сотрудников своей работой.

Подход DevOps требует новых культурных и управлеченческих норм, а также изменений в технических методиках и в архитектуре. Для этого нужно тесное сотрудничество руководства компании, подразделений управления продуктами, разработки, тестирования, эксплуатации, информационной безопасности и даже маркетинга, где часто зарождаются многие перспективные идеи. Когда все команды работают вместе, мы можем создать безопасную систему работы: небольшие группы быстро и самостоятельно пишут и проверяют код, его можно безопасно развертывать в эксплуатацию. В результате максимизируются продуктивность разработчиков, удовлетворенность сотрудников работой и способность компаний накапливать опыт и отвоевывать рынок у конкурентов.

Цель этой книги — подробная кодификация принципов и методик DevOps, чтобы другие компании могли добиться тех же потрясающих результатов, что и сообщество DevOps. Мы надеемся, что у вас получится ускорить распространение идей DevOps, успешно воплотить их в жизнь, сократив необходимые затраты сил и энергии на эти преобразования.

Мы знаем, как опасно откладывать улучшения и сосредоточивать внимание на срочной работе. Мы знаем, как сложно менять привычный режим повседневной работы. Кроме того, понимаем, какого риска и каких усилий требует введение в компании новых методов работы. Недаром многие воспринимают DevOps как очередной новомодный подход-однодневку, существующий лишь до появления чего-то еще нового.

Мы утверждаем: DevOps выводит организацию рабочих процессов компании на качественно новый уровень, точно так же как в 1980-е гг. методы бережливого производства совершили революцию в промышленности. Рынок захватят те, кто примет подход DevOps, за счет тех, кто от него откажется. Они создадут полные энергии, ориентированные на обучение компаний, и те превзойдут конкурентов и в производительности, и в способности к инновациям.

По этим причинам освоение DevOps — настоятельная необходимость не только с технологической точки зрения, но и с точки зрения управления компанией. Резюмируя вышесказанное, можно сделать вывод: DevOps важен и применим в любой компании, желающей увеличить поток планируемой работы посредством технологической системы и в то же время поддерживающей качество, надежность и безопасность сервисов для клиентов.

Наш призыв к действию таков: какая бы роль в компании у вас ни была, начинайте искать вокруг тех, кто хотел бы что-то изменить в рабочем процессе. Покажите эту книгу остальным и создайте объединение единомышленников, чтобы вырваться из порочного круга плохих методик. Попросите начальников поддержать ваши начинания или, что еще лучше, возглавьте инициативу и сами найдите для нее бюджет.

Наконец, раз уж вы дочитали до этого места, откроем страшную тайну. Мы разбирали практические примеры. Так вот, после демонстрации потрясающих результатов новых методик инициаторы получили повышение. Однако в некоторых случаях руководство менялось, и новаторы были вынуждены уйти, а компании возвращались к старым методам организации процессов.

Мы верим, что важно не падать духом. Те, кто занимается преобразованием привычных способов работы, с самого начала понимали: их инициативы с большой долей вероятности могут провалиться. Но они все равно действовали. Возможно, самым важным результатом становилось то, что они показали, каких результатов можно добиться. Инновации без риска невозможны, и, если вы не расстроили хотя бы одного начальника, видимо, вы недостаточно сильно стараетесь. Не позволяйте

иммунной системе компании отвлечь вас от вашей задачи. Как любит говорить Джесс Роббинс, «мастер аварий» компании Amazon, «не боритесь с глупостью, делайте больше крутого».

Методология DevOps приносит пользу всем, будь то разработчики, инженеры эксплуатации, тестировщики, инженеры информационной безопасности, заказчики или клиенты. Она возвращает радость в разработку важных сервисов, сокращая число авральных марафонов. Она делает условия работы более человечными, и вы можете спокойно проводить ваши выходные и праздники вместе с любимыми. Благодаря ей разные команды могут работать вместе, чтобы выживать, учиться, процветать, радовать клиентов и приносить пользу компании.

Мы искренне надеемся, что Руководство по DevOps поможет вам достичь этих целей.

## Приложения

Мы считаем, что DevOps сильно выигрывает от взаимодействия разных направлений менеджмента, усиливающих друг друга и создающих мощную систему, способную изменить традиционные подходы к разработке и поставке ИТ-продуктов и сервисов.

Джон Уиллис назвал этот процесс «конвергенцией DevOps». Подходы к управлению, ставшие предками DevOps, описаны ниже в порядке появления (отметим, что это не подробные описания, а скорее, заметки, призванные показать развитие мысли и неочевидные связи между направлениями. Они в итоге привели к созданию DevOps).

Бережливое производство возникло в 1980-х гг. как попытка формализовать производственную систему компании Toyota и популяризовать такие методики, как систематизирование потока ценности, канбан-доски и всеобщий уход за оборудованием.

Два основополагающих принципа бережливого производства — глубокая вера в то, что время производственного цикла (то есть время, затраченное на преобразование исходных материалов в готовую продукцию) — лучший показатель качества работы, удовлетворенности клиентов и сотрудников, а также того, что одним из главных факторов сокращения времени производственного цикла были небольшие размеры партии. Идеалом был «поток единичных изделий» (то есть поток « $1 \times 1$ »: одна единица исходных материалов — одна единица готовой продукции).

Принципы бережливого производства — систематическое мышление, формулировка четкой цели, использование научного подхода, создание потока и вытягивания вместо выталкивания, изначальное обеспечение качества, управление на основе скромности и уважение к каждому человеку. Все это сосредоточено на создании ценности для клиента.

Agile-манифест был создан в 2001 г. семнадцатью ведущими мыслителями в области разработки ПО. Их целью было преобразование таких «неглубоких» методов, как DP и DSDM, в более широкую систему, куда можно было бы включить более масштабные подходы к разработке, такие как каскадная модель (Waterfall Model) или унифицированный процесс разработки (Rational Unified Process).

Ключевой принцип — «частая поставка работающего программного обеспечения, от нескольких недель до нескольких месяцев, чем быстрее — тем лучше». Два других важных принципа гибкой разработки — небольшие, целеустремленные команды, работающие по модели управления с высоким уровнем доверия, и предпочтение небольшим объемам работы. С гибкой методологией разработки также связаны такие наборы инструментов и методики, как Scrum, Stand-up и так далее.

Конференция Velocity, впервые проведенная в 2007 г., была создана Стивом Судерсом, Джоном Оллспоу и Джессом Робинсоном как место, где инженеры эксплуатации и производительности web-сервисов могли бы чувствовать себя как дома. На конференции в 2009 г. Джон Оллспоу и Пол Хэммонд представили основополагающий доклад 10 Deploys per Day: Dev and Ops Cooperation at Flickr («Десять развертываний в день: сотрудничество разработки и эксплуатации во Flickr»).

В 2008 г. на конференции гибкой методологии разработки в Торонто Патрик Дюбуа и Эндрю Шейфер провели сессию единомышленников, посвященную применению принципов гибкой разработки к инфраструктуре, а не только к коду. У них быстро появились последователи, например Джон Уиллис. Позже Дюбуа был настолько впечатлен докладом Оллспоу и Хэммонда 10 Deploys per Day: Dev and Ops Cooperation at Flickr, что провел первую конференцию DevOpsDays в Генте, Бельгия, в 2009 г., где впервые прозвучал термин DevOps.

Развивая идеи непрерывной сборки, тестирования и интеграции разработки ПО, Джез Хамбл и Дэвид Фарли придумали концепцию непрерывной поставки. В нее входит понятие «конвейер развертывания». Суть в том, что код и инфраструктура всегда готовы к развертыванию и что в развертывание всегда уходит весь код в основной ветке.

Эта идея впервые была представлена на конференции Agile в 2006 г., кроме того, независимо описана Тимом Фитцем в блоге в статье под названием Continuous Deployment («Непрерывное развертывание»).

В 2009 г. Майк Ротер написал книгу Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results. В ней был обобщен его двадцатилетний опыт по анализу и приведению в систему причинных механизмов производственной системы компании Toyota. В книге описываются «невидимые управленческие практики и способы мышления, определяющие успех Toyota в непрерывном улучшении процессов и в адаптации... и то, как другие компании используют похожие методики и способы мышления в своей работе».

Его вывод: сообщество упустило самую важную методику бережливого производства, описанную как «ката улучшения». Ротер объясняет: в каждой организации есть свои привычные способы

организации работы, и важнейший фактор в Toyota — то, они сделали улучшение повседневным процессом и встроили его в ежедневную работу всех сотрудников. Тойота Ката декларирует постепенный, повторяющийся, научный подход к решению проблем. Главная цель — достижение целей компании.

В 2011 г. Эрик Рис написал книгу *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, обобщив опыт работы в IMVU, стартапе в Кремниевой долине, организованном в соответствии с принципами Стива Бланка, которые описаны в книге *The Four Steps to the Eriphany*, и использовавшем методики непрерывного развертывания. Эрик Рис также сформулировал такие важные методики и понятия, как минимально жизнеспособный продукт, цикл «создать — измерить — сделать выводы» и большое число шаблонов непрерывной поставки.

В 2013 г. Джек Готельф написал книгу *Lean UX: Applying Lean Principles to Improve User Experience*. В ней описывалось, как улучшить «туманный front-end» и как собственники продукта могут формулировать гипотезы, проводить эксперименты и убеждаться в верности этих гипотез до того, как вкладывать время и ресурсы в возможные компоненты функциональности. Благодаря Lean UX у нас теперь есть инструменты, чтобы полностью оптимизировать поток между выдвижением гипотез, разработкой функциональности, тестированием, развертыванием и предоставлением сервиса пользователям.

В 2011 г. Джошуа Кормэн, Дэвид Райс и Джек Уильямс изучили кажущуюся бесполезность обеспечения безопасности приложений и среды на поздних этапах жизненного цикла. В ответ на подобные заявления они создали принцип прочной разработки (*Rugged Computing*), благодаря чему формируется понимание таких нефункциональных требований, как стабильность, масштабируемость, доступность, выживаемость, устойчивость, безопасность, поддерживаемость, управляемость и защищенность.

Из-за высоких темпов выпуска релизов система DevOps может оказывать существенное давление на тестировщиков и службу информационной безопасности: когда темпы развертывания меняются с ежемесячных или ежеквартальных на сотни или тысячи развертываний в день, привычные двухнедельные сроки работы этих служб уже не соответствуют изменившейся действительности. Принцип прочной разработки показывает, что текущий подход к борьбе с уязвимыми местами промышленных комплексов, описанный в большинстве нынешних программ информационной безопасности, устарел.

В теории ограничений широко используется метод облака ключевых конфликтов (*core conflict cloud*, С). Вот пример облака конфликтов для ИТ-компаний (рис. 46).



Рис. 46. Ключевой хронический конфликт в любой ИТ-организации

В 1980-е гг. был распространен ключевой хронический конфликт в промышленности. Любой руководитель завода имел две важные цели: сохранить уровень продаж и сократить издержки. Чтобы поддержать уровень продаж, руководству требовалось увеличить складские площади, чтобы можно было всегда удовлетворить потребительский спрос. И тут возникала проблема.

Ведь, с другой стороны, руководство должно было сократить складские площади, чтобы снизить издержки и чтобы капитал в виде неотгруженной продукции, за которую нельзя получить деньги, не выходил из оборота.

Конфликт удалось разрешить с помощью принципов бережливого производства: уменьшить

размеры партий и объемы незавершенного производства, сократить и усилить петли обратной связи. В результате производительность заводов, качество продуктов и удовлетворенность клиентов резко увеличились.

Принципы шаблонов работы DevOps — те же самые, совершившие переворот в промышленном производстве. Они позволяют оптимизировать поток создания ценности и трансформируют потребности бизнеса в возможности и сервисы, удовлетворяющие нужды клиентов.

Нисходящая спираль, описанная в книге The Phoenix Project, может быть представлена в виде таблицы:

Таблица 4. Нисходящая спираль

В эксплуатации видят, что...	При разработке видят, что...
В хрупких приложениях часто происходят сбои	В хрупких приложениях часто происходят сбои
Нужно много времени, чтобы понять, какой же бит был инвертирован	В очереди есть более срочные задачи с жестким дедлайном
К поискам проблем в процессах подключаются менеджеры по продажам	В эксплуатацию поступает еще более хрупкий (менее безопасный) код
Нужно слишком много времени, чтобы восстановить сервис	Число релизов со все более неустойчивой установкой растет
Слишком много ликвидации срочных кризисов, незапланированной работы	Циклы релизов удлиняются, чтобы уменьшить издержки развертывания
Срочная доработка и восстановление системы защиты данных сервиса	Сбои в масштабных развертываниях все труднее диагностировать
Нет времени, чтобы закончить запланированную работу над проектом	У самых опытных и занятых сотрудников эксплуатации все меньше времени на исправление глубинных проблем процессов
Разочарованные клиенты уходят	Очередь необходимых работ, которые помогли бы бизнесу преуспеть, все растет и растет
Доля рынка падает	Число конфликтов между разработкой, проектированием и эксплуатацией растет
Бизнес не выполняет свои обязательства	
Бизнес берет на себя еще более крупные обязательства	

Проблемы долгого ожидания в очереди обостряются, когда проект много раз передается от инстанции к инстанции, поскольку именно здесь очереди и возникают. На рис. 47 показано время ожидания как функция занятости ресурса производственного участка. Асимптотическая кривая демонстрирует, почему простое изменение, требующее всего-то полчаса, часто отнимает недели для завершения: часто узкие места создают конкретные инженеры и производственные участки с высоким коэффициентом занятости. Когда какой-нибудь производственный участок приближается

к 100 %-ной загруженности, любая другая работа неизбежно застrevает, пока кто-нибудь наконец не займется пробкой.

На рис. 47 по оси X отложен процент занятости данного ресурса на производственном участке, по оси Y — примерное время ожидания (или, что более точно, длина очереди). Форма кривой демонстрирует: когда загруженность ресурса превышает 80 %, время ожидания зашкаливает.

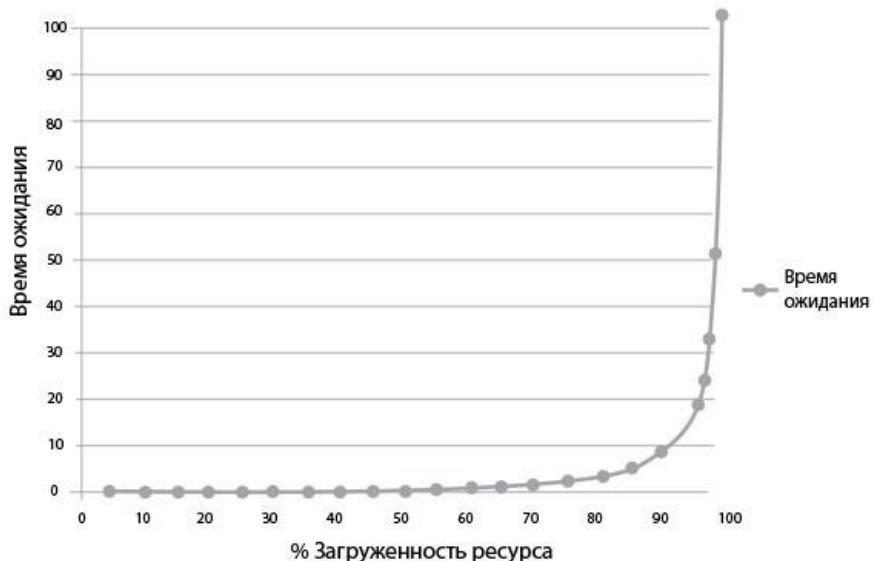


Рис. 47. Размер очереди ожидания и время ожидания как функция процента загруженности (источник: Ким, Бер и Спаффорд, The Phoenix Project, ePub edition, 557)

В книге The Phoenix Project рассказывается, как Билл Шинн и его команда осознали губительное влияние этой зависимости на среднее время подтверждения кода, отправленного в центр управления проектами:

«Эрик рассказал мне во время MRP-8, как время ожидания зависит от использования ресурсов. По его словам, время ожидания — это процент времени занятости, поделенное на процент свободного времени. Другими словами, если ресурс занят половину времени, другую половину он свободен. Время ожидания тогда — 50 % делить на 50 %, то есть одна единица. Будем считать ее равной одному часу.

Значит, в среднем наша задача будет ждать в очереди один час, прежде чем сможет выполниться.

С другой стороны, если ресурс занят 90 % времени, время ожидания — 90 %, деленное на 10 %, или девять часов. Другими словами, наша задача будет ждать в очереди в девять раз дольше, чем если бы ресурс был занят наполовину.

Я завершаю мысль. Все это значит, что при условии, что передача задачи между инстанциями у нас происходит семь раз и что каждый из ресурсов занят 90 % времени, задачи проведут в очереди в сумме девять часов умножить на семь шагов.

“Что? Шестьдесят три часа только на ожидание в очереди? — недоверчиво говорит Уэс. — Невозможно!”

Патти с усмешкой отвечает: “О, ну конечно. Это ведь всего лишь тридцать секунд ввода команды, да?”»

Билл и команда отдают себе отчет, что «простое задание на полчаса» на самом деле требует семи передач по разным инстанциям (например, командам по серверам, по сетевым соединениям, по базам данных, команде виртуализации и, конечно же, Бренту, «звездному» инженеру).

При условии, что все производственные участки были заняты 90 % времени, на рисунке видно, что среднее время ожидания на каждом участке — девять часов. А поскольку задача должна пройти через семь участков, суммарное время ожидания в семь раз больше: это целых шестьдесят три часа.

Другими словами, суммарная доля *полезного времени* (также известного как длительность процесса) составляла всего 0,16 % от затраченного времени (тридцать минут, поделенных на шестьдесят три часа). Это значит, что 99,8 % всего времени задача бессмысленно провела в

очереди ожидания.

Десятилетия изучения сложных систем свидетельствуют о том, что контрмеры часто основываются на нескольких мифах. Они описаны в книге Дени Беснара и Эрика Холлнагела *Some Myths about Industrial Safety*.

- **Миф 1:** «Ошибки людей — главная причина неполадок и инцидентов».
- **Миф 2:** «Системы в безопасности, если следовать инструкции».
- **Миф 3:** «Системы можно улучшить с помощью барьеров и ограничений: чем выше уровень защиты, тем выше безопасность».
- **Миф 4:** «Анализ сбоев может выявить истинную причину сбоя».
- **Миф 5:** «Расследование сбоя — логичное и рациональное определение его причин, основанное на фактах».
- **Миф 6:** «Безопасность всегда имеет высший приоритет, она никогда не оказывается под угрозой».

Разница между мифами и реальностью показана ниже (табл. 5).

Таблица 5. Две истории

Миф	Реальность
Причиной сбоев кажутся ошибки людей	Ошибки людей — отражение более глубоких системных уязвимых мест компаний
Чтобы описать сбой, достаточно просто сказать, что нужно было в этой ситуации сделать	Описание правильного поведения ничего не говорит о том, почему в тот момент ошибка казалась правильным поступком
Если просто сказать сотрудникам быть осторожнее, то все проблемы исчезнут	Компания может усилить безопасность и надежность только с помощью постоянного поиска и выявления уязвимых мест

Многие спрашивают: как работа вообще выполняется, если за шнур-андон дергают больше 5000 раз в день? Если быть точным, не каждое срабатывание системы андон приводит к остановке конвейера. Когда кто-нибудь дергает шнур, у руководителя группы, отвечающего за конкретный производственный участок, есть 50 секунд, чтобы разобраться. Если за это время проблема не была решена, собираемый автомобиль пересечет нарисованную на полу линию, и конвейер остановится.

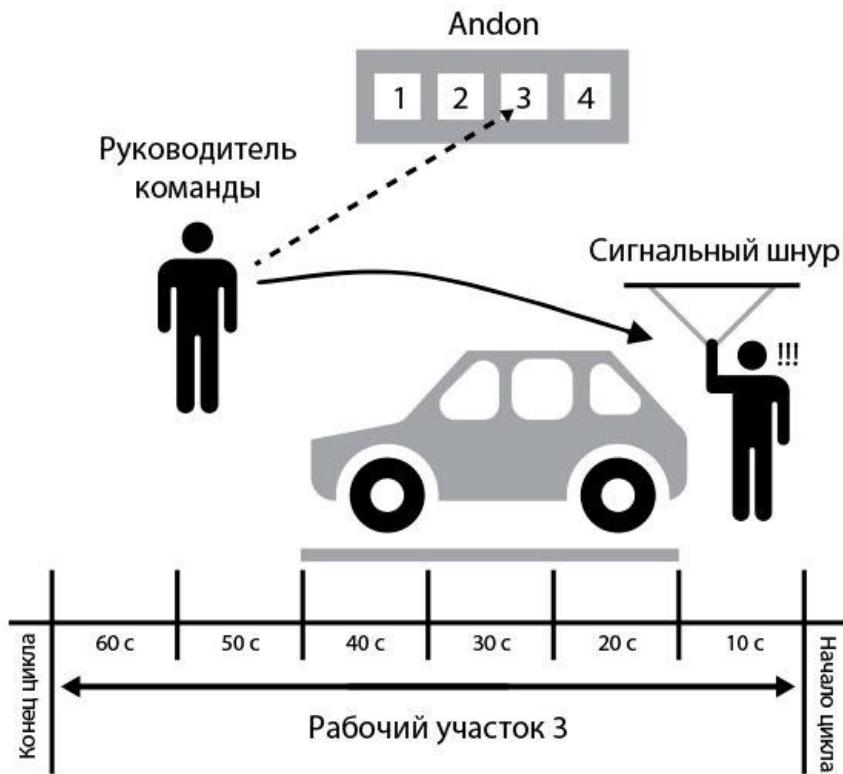


Рис. 48. Шнур-андон компании Тойота

На данный момент, чтобы встроить сложное коммерческое готовое программное обеспечение (commercial off-the-shelf, COTS; например SAP, IBM WebSphere, Oracle WebLogic) в систему контроля версий, придется, возможно, прекратить использовать графические инструменты установки, предоставляемые поставщиком. Для этого нужно разобраться, что именно делает установщик. Возможно, потребуется сделать установку на чистый образ сервера, сравнить файловые системы и ввести добавленные файлы в систему контроля версий. Файлы, не меняющиеся в зависимости от среды, помещаются в одно место («базовая установка»), тогда как специфичные для разных сред помещаются в отдельные папки («тест» или «эксплуатация»). Так установка программ становится просто операцией в системе контроля версий. Прозрачность, повторяемость и скорость операций улучшаются.

Также, вероятно, придется изменить настройки конфигурации приложений, чтобы они были в системе контроля версий. Например, можно преобразовать конфигурации приложений, хранящихся в базе данных, в XML-файлы, или наоборот.

Ниже приведен простой план совещания по разбору для послеаварийной ретроспективы.

- В самом начале руководитель встречи или координатор произносит небольшую вступительную речь, подчеркивая, что сегодня никто не будет искать виноватых, сосредоточиваясь на прошедших событиях или рассуждать о том, что могло бы или должно было быть. Координатор может прочитать главную директиву разбора ошибок (Retrospective Prime Directive) с сайта .

Кроме того, координатор должен напомнить всем, что у контроллер должен быть конкретный исполнитель. Если по окончании совещания контроллер не оказывается приоритетной, то это не контроллера (так делается, чтобы не создавать длинный список хороших идей, которые никогда не воплотятся в жизнь).

- Участники встречи должны составить единую картину того, в каком порядке происходили события во время сбоя: кто и когда обнаружил неполадку, как она была обнаружена (например, с помощью автоматического мониторинга, контроля вручную, письма клиента), когда работоспособность сервиса была восстановлена и так далее. В последовательную цепочку событий также нужно внести все внешние коммуникации во время инцидента.

Используя выражение «цепочка событий», мы формируем в воображении образ линейной последовательности шагов: как формировалось понимание проблемы и как мы в итоге ее исправили. На самом деле, особенно в сложных системах, к сбою приводят много разных событий, и нужно вносить исправления по нескольким путям одновременно. На этом шаге следует отследить все события и все мнения участников и по возможности выдвинуть гипотезы о причинно-

следственных связях.

- Далее команда создает список всех факторов, приведших к инциденту: и человеческих, и технических. Потом их можно распределить по категориям, например «проектировочное решение», «восстановление», «фиксация наличия проблемы» и так далее. Команда может использовать такие методики, как мозговой штурм и «бесконечные «как»», чтобы вскрыть более глубокие причины проблемы, если в этом есть необходимость. При этом все точки зрения должны восприниматься уважительно — никто не должен возражать или спорить с реальностью фактора, предложенного кем-то другим. Очень важно, чтобы координатор выделил достаточно времени на эту часть совещания и чтобы команда не пыталась свести все к одной-двум «главным причинам».
- На следующем этапе участники совещания должны определиться со списком корректирующих действий, которые нужно будет выполнить как можно быстрее. Чтобы составить список, полезно устроить мозговой штурм. По итогам необходимо выбрать наилучшие действия для предотвращения таких ошибок в будущем или хотя бы для их более быстрого обнаружения. Туда можно включить и другие способы улучшить рабочие системы.

Наша цель — определить наименьшее число небольших шагов для достижения желаемых результатов, в противоположность глобальным изменениям, отнимающим больше времени и замедляющим введение других необходимых изменений.

Также нужно составить другой список — менее приоритетных идей — и назначить ответственного за него. Если в будущем возникнут похожие проблемы, список может послужить отправной точкой возможных решений.

Участники совещания должны определиться с характеристиками инцидентов и их влиянием на организацию. Например, сбои можно характеризовать следующими показателями.

**Тяжесть инцидента:** насколько серьезной была проблема? Этот показатель непосредственно связан с влиянием на сервис и на клиентов.

**Время простоя:** как долго клиенты не могли пользоваться сервисом?

**Время обнаружения:** сколько времени потребовалось на то, чтобы заметить, что есть проблема?

**Время устранения проблемы:** сколько времени потребовалось на то, чтобы восстановить работу сервиса после того, как мы обнаружили сбой?

Бетани Макри из компании Etsy отмечает: «Отсутствие обвинений на совещаниях не означает, что никто не берет на себя ответственность. Но мы хотим понять, какие обстоятельства привели к тому, что человек совершил ошибку, каков был широкий контекст. Главная идея в том, что, исключив ответственность, вы устраняете страх; устранив страх, допускаете честность; тогда честность дает возможность предотвратить сбой».

После масштабного сбоя AWS EAST 2011 г. в компании Netflix активно обсуждали, как сделать, чтобы системы сами справлялись с неполадками. Из этих дискуссий вырос инструмент под названием Chaos Monkey.

С тех пор этот сервис развился в целый набор инструментов, известный как «Обезьяньая армия Netflix» и призванный симулировать разные уровни сбоев.

• **Горилла Хаоса (Chaos Gorilla):** симулирует отказ целой зоны доступности AWS.

• **Хаос-Конг (Chaos Kong):** симулирует отказ целого региона AWS, например североамериканского или европейского.

Среди других бойцов Обезьяньей армии можно отметить следующих.

• **Обезьяна Задержек (Latency Monkey):** создает искусственные задержки или остановку работы на уровне связи «клиент — сервер», соответствующей ограничениям REST, чтобы симулировать плавный отказ сервиса и проконтролировать, что зависимые сервисы отвечают на это надлежащим образом.

• **Обезьяна Согласованности (Conformity Monkey):** находит и выводит из работы инстансы AWS, не соответствующие стандартным значениям (например, когда инстансы не принадлежат к автоматически масштабируемой группе или когда в каталоге сервиса не указан адрес электронной

почты ответственного инженера).

• **Обезьяна Доктор (Doctor Monkey)**: просматривает результаты проверок работоспособности каждого инстанса, выявляет больные инстансы и проактивно отключает их, если ответственные за них инженеры не устраняют проблему вовремя.

• **Обезьяна Уборщик (Janitor Monkey)**: следит за тем, чтобы в облачной среде не было мусора и хлама; ищет неиспользуемые ресурсы и избавляется от них.

• **Обезьяна Безопасности (Security Monkey)**: расширение Обезьяны Согласованности; ищет и выводит из работы инстансы с нарушениями безопасности и уязвимыми местами, например неверно настроенные группы безопасности AWS.

Ленни Рачицки о преимуществах Transperant Uptime («прозрачности работы сервисов для клиентов»):

1. Снижаются издержки на поддержание сервисов, так как пользователи сами могут идентифицировать проблемы ваших систем без звонков или писем в службу поддержки. Пользователям больше не приходится угадывать, локальные у них проблемы или глобальные, они могут быстрее определить причины сбоя и сообщают о неполадках, уже зная существование проблемы.
2. В противоположность общению один на один по электронной почте контакт с пользователями во время выхода сервиса из строя становится продуктивнее, так как благодаря открытости интернета можно обращаться сразу к большой аудитории. Вы тратите меньше времени на воспроизведение одной и той же информации и можете сосредоточиться на решении проблемы.
3. Создается единый пункт для пользователей, куда они могут обратиться, когда сервис выходит из строя. Вы экономите их время. Иначе они потратили бы его на долгий поиск по форумам или вашему блогу.
4. Доверие — краеугольный камень перехода на модель SaaS (Software as a Service, программное обеспечение как услуга). Ваши клиенты ставят свой бизнес в зависимость от вашего сервиса или платформы. И текущим, и потенциальным клиентам нужна уверенность в вашем сервисе. Им нужно знать, что они не останутся без помощи, если у вас возникнут проблемы. Предоставлять информацию о форс-мажорах в режиме реального времени — лучший способ строить доверительные отношения. Больше вы не оставите клиентов в одиночестве без информации о текущей ситуации.
5. Всего лишь вопрос времени, когда же все серьезные SaaS-провайдеры начнут публиковать данные о работоспособности своих сервисов. Пользователи сами потребуют этого.

## **Дополнительная литература**

Многие проблемы IT-организаций описаны в первой половине книги *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*, авторы Джин Ким, Кевин Бер и Джордж Спаффорд.

По ссылке можно услышать рассказ Пола О'Нила об опыте, приобретенном в должности CEO компании Alcoa; там также говорится и о расследовании смерти работавшего в компании подростка, в котором О'Нил принял участие: .

Если вы хотите больше узнать о картировании потока ценности, посмотрите книгу Карен Мартин и Майка Остерлинга *Value Stream Mapping: How to Visualize Work and Align Leadership for Organizational Transformation*.

Более подробную информацию об объектно-реляционных отображениях можно найти в статье на сайте Stack Overflow: .

Информативные статьи для начинающих о методиках гибкой методологии разработки и о том, как использовать их в эксплуатации, можно найти в блоге Agile Admin: .

Больше информации об архитектурном проектировании для быстрых сборок можно найти в блоге Даниэля Уортингтона-Бодарта *Crazy Fast Build Times (or When 10 Seconds Starts to Make You Nervous)*: .

Чтобы полнее ознакомиться с тестированием производительности и процессом выпуска релизов в Facebook, посмотрите презентацию Чака Росси *The Facebook Release Process* .

Гораздо больше вариантов теневого запуска вы найдете в главе 8 книги Томаса Лимончелли, Страты Чалупа и Кристины Хоган *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2*.

Отличное обсуждение о переключателях элементов функциональности (feature toggles) есть вот здесь: .

Релизы более детально обсуждаются в книгах *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2* Томаса Лимончелли, Страты Чалупа и Кристины Хоган; *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* Джеза Хамбла и Дэвида Фарли; *Release It! Design and Deploy Production-Ready Software* Майкла Нейгарда.

Описание шаблона автоматического прерывателя вы можете найти здесь: .

Чтобы лучше познакомиться с тем, что такое цена промедления, посмотрите книгу Дональда Рейнерсена *The Principles of Product Development Flow: Second Generation Lean Product Development*.

Более подробное обсуждение того, как сервис Amazon S3 работает со сбоями, находится здесь: .

Отличным путеводителем по проведению исследований поведения пользователей может послужить книга Джоша Сайдена *Lean UX: Applying Lean Principles to Improve User Experience*.

На сайте *Which Test Won?* приведены сотни примеров реальных A/B-тестов, и можно попробовать угадать, какой вариант оказался предпочтительнее, чтобы еще раз убедиться: без настоящих тестов мы просто играем в угадайку. Сайт находится по адресу: .

Список архитектурных шаблонов можно найти в книге Майкла Нейгарда *Release It! Design and Deploy Production-Ready Software*.

Пример заметок со встречи для послеаварийной ретроспективы компании Chef можно найти здесь: . Видео совещания находится здесь: .

Расписание предстоящих конференций DevOpsDays можно найти на сайте *DevOpsDays*: . Инструкции по организации новой конференции DevOpsDays находятся на странице *DevOpsDay Organizing Guide*: .

Более подробно об инструментах по управлению секретной информацией можно узнать из поста Ноа Кантровица *Secrets Management and Chef* в его блоге .

Джеймс Уикетт и Гарет Рашгров разместили все примеры безопасных конвейеров на сайте GitHub: .

Сайт *The National Vulnerability Database* и каналы данных XML можно найти по адресу: .

Конкретный сценарий по интеграции Puppet, ThoughtWorks' Go и Mingle (приложение по управлению проектами) можно найти в посте блога Puppet Labs, написанном Эндрю Каннингемом и Эндрю Майерсом и отредактированном Джезом Хамблом: .

Подготовка и прохождение аудита на соответствие требованиям более подробно разобраны в презентации Джейсона Чана 2015 г. SEC310: Splitting the Check on Compliance and Security: Keeping Developers and Auditors Happy in the Cloud: .

История о том, как Джез Хамбл и Дэвид Фарли преобразовали настройки конфигурации приложения для Oracle WebLogic, описывается в книге Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Мирко Херинг описал более общий подход к этому процессу здесь: .

Примерный список эксплуатационных требований DevOps можно найти здесь: .

## **МИФ Бизнес**

Все книги по бизнесу и маркетингу: ,

Узнавай первым о новых книгах, скидках и подарках из нашей рассылки



**Над книгой работали**

Издано при поддержке компании «Неофлекс»

Главный редактор *Артем Степанов*

Ответственный редактор *Наталья Хоренко*

Литературный редактор *Вера Калмыкова*

Арт-директор *Алексей Богомолов*

Дизайн обложки

Верстка *Елена Бреге*

Корректоры *Лев Зелексон, Елена Бреге*

**ООО «Манн, Иванов и Фербер»**

Электронная версия книги подготовлена компанией Webkniga.ru, 2018

## **Примечания**

# 1

Акроним от англ. development и operations — методология разработки программного обеспечения, нацеленная на активное взаимодействие и интеграцию специалистов по разработке и специалистов по ИТ-обслуживанию. *Прим. перев.*

*Ким Д., Бер К., Слаффорд Дж.* Проект «Феникс». Роман о том, как DevOps меняет бизнес к лучшему. М.: Эксмо, 2015. Прим. перев.

Техника «Blue-Green разворачивания» — стратегия установки ПО, базирующаяся на двух идентичных инсталляциях промышленной системы, одна из которых активна, и возможно мгновенное переключение между ними. Одна из них условно называется синей, ее копия же называется зеленой. *Прим. перев.*

Хамбл Д., Фарли Д. Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ... М.: Вильямс, 2011. Прим. перев.

Институт инженеров электротехники и электроники (от англ. Institute of Electrical and Electronics Engineers) — международная некоммерческая ассоциация специалистов в области техники, мировой лидер в области разработки стандартов по радиоэлектронике, электротехнике и аппаратному обеспечению вычислительных систем и сетей. *Прим. перев.*

Continuous Integration / Continuous Deployment — непрерывная интеграция и непрерывное развертывание. *Прим. ред.*

Это всего лишь небольшой пример проблем, встречающихся в типичной ИТ-организации. *Прим. авт.*

Коммодитизация — превращение изделий в обезличенный товар. *Прим. перев.*

Термин из области разработки ПО: как только появляются изменения в программном коде, зачастую возникает необходимость сделать связанные с ними изменения в других частях кода или документации. Эти необходимые, но незавершенные изменения считаются долгом. Он должен быть погашен в определенный момент в будущем. *Прим. перев.*

В мире промышленного производства существует похожий корневой, хронический конфликт. Его суть в необходимости обеспечивать своевременные поставки клиентам и управление затратами. Как этот конфликт был преодолен, рассказывается в . *Прим. авт.*

В 2013 году в европейском банке HSBC трудилось больше разработчиков ПО, чем в компании Google. *Прим. авт.*

Пока не будем устраивать дискуссий, как должна финансироваться разработка ПО, как «проект» или как «продукт». Это мы обсудим ниже. *Прим. авт.*

Например, Вернон Ричардсон с коллегами опубликовал следующие поразительные результаты. Они изучили отчеты 184 публичных корпораций по форме 10-K и разделили их на три группы: а) фирмы с нехваткой материальных ресурсов и с нечеткой работой IT-подразделений; б) фирмы с нехваткой материальных ресурсов и с четкой работой IT-подразделений; в) «чистые фирмы» без нехватки материальных ресурсов. В фирмах из группы А текучка кадров среди руководителей высшего звена была в восемь раз выше, чем в фирмах из группы В, а в фирмах из группы Б — только в четыре раза. Ясно, что работа IT-структур оказывается гораздо более важной, чем принято считать. *Прим. авт.*

По данным индекса чистой лояльности сотрудников (employee Net Promoter Score — eNPS). Это очень значимое открытие, поскольку исследование доказало, что «компании, где работники имеют высокую удовлетворенность, показали в 2,5 раза более высокий рост доходов, чем компании с низкой удовлетворенностью работников. И [торгующиеся на бирже акции] компании с высоким уровнем удовлетворенности показали за период с 1997 по 2011 г. рост цены, втрое превышающий рост биржевых индексов». *Прим. авт.*

М.: Символ-Плюс, 2010. *Прим. перев.*

## **16**

Показаны данные только по тем организациям, которые выполняют хотя бы одно развертывание в день. *Прим. авт.*

Другой выдающийся пример — компания Amazon. В 2011 г. она выполняла около 7000 развертываний в день. В 2015 г. — уже 130 000. *Прим. авт.*

Издана: М.: Альпина Диджитал, 2014. *Прим. перев.*

DevOps также расширяет и усовершенствует принцип инфраструктура как код, впервые предложенный Марком Берджессом, Люком Канисом и Адамом Джекобом. При использовании этого принципа работа эксплуатации автоматизирована и трактуется как разработка кода приложений, так что все современные методики разработки могут быть применены ко всему потоку разработки и управления. Это увеличивает возможности ускорения потока разработки, включая непрерывную интеграцию (придумана Греди Бучем как один из 12 ключевых методов «экстремального программирования»), непрерывную поставку (впервые введена Джезом Хамблом и Дэвидом Фарли) и непрерывное развертывание (созданное компаниями Etsy и Wealthfront, а также Эриком Рисом из IMVU). *Прим. авт.*

Издана: СПб.: Питер, 2014. *Прим. перев.*

Ката — формализованная последовательность движений, связанных принципами ведения поединка с воображаемым противником или группой противников. По сути, является квинтэссенцией техники конкретного стиля боевых искусств. *Прим. ред.*

Здесь и далее слово «инженер» означает любого человека, работающего в команде, производящей поток ценности, а не только разработчиков. *Прим. авт.*

По правде говоря, при использовании таких методик, как разработка через тестирование, тестирование может начинаться еще до того, как будет написана первая строка кода. *Прим. авт.*

В этой книге термин «время производства» будет использоваться по той же причине, о которой говорили Карен Мартин и Майк Остерлинг: «чтобы минимизировать возможную путаницу, мы избегаем использовать термин “время цикла”, поскольку он имеет несколько значений, в том числе синонимичное для времени производства и темпа или частоты выдачи результатов, и это только некоторые». *Прим. авт.*

Тайити Оно сравнил введение лимитов на НзП с отводом воды из реки производства, чтобы обнажить все проблемы, препятствующие свободному течению. *Прим. авт.*

Также это называют «единичный размер партии» или «поток  $1 \times 1$ », потому что и размер партии, и размер НзП равны единице. *Прим. авт.*

Издана: М.: Альпина Паблишер, 2016. *Прим. перев.*

Издана: М.: Вильямс, 2010. *Прим. перев.*

Хотя Мэри и Том Поппендики не включили геройство в категорию потерь, мы это сделали в связи с высоким уровнем распространности таких ситуаций, особенно при эксплуатации в случае систем, обслуживающих много клиентов.

Спир расширил рамки своего исследования, чтобы объяснить, почему и другие компании оставались успешными в течение долгого времени, в частности сеть снабжения компаний Toyota, Alcoa, программа военно-морских сил США по разработке атомных двигательных систем.

Издана: М.: Олимп-Бизнес, 2003. *Прим. перев.*

На некоторых заводах Toyota вместо шнура стали использовать кнопку Andon. *Прим. авт.*

В XVIII веке британское правительство продемонстрировало пример иерархически организованной бюрократической системы командования и управления, оказавшейся чрезвычайно неэффективной. В то время Джордзия еще была колонией, и хотя британское правительство находилось в 5000 километров и не имело сведений из первых рук о расположении земель, каменистости почв, топографии, доступности водных ресурсов и других условий, оно попыталось спланировать сельскохозяйственную экономику Джордзии. Результаты оказались плачевными: благосостояние и население Джордзии стали самыми низкими среди всех 13 колоний. *Прим. авт.*

Шаблон «позор тебе, NN» — часть теории плохого яблока, раскритикованной Сидни Деккером и активно обсуждавшейся в его книге *The Field Guide to Understanding Human Error*. *Прим. авт.*

Генеральный директор. *Прим. ред.*

Это удивительно, познавательно и действительно демонстрирует убежденность и страсть, с которыми О'Нил относился к понятию моральной ответственности руководителей за создание безопасных условий труда. *Прим. авт.*

Руководители несут ответственность за разработку и эксплуатацию процессов на более высоком уровне обобщения, где рядовые сотрудники не имеют перспективного видения и достаточной власти. *Прим. авт.*

Организации, проектирующие системы... производят их, копируя структуры коммуникации, сложившиеся в этих организациях (Конвей, 1968). *Прим. перев.*

Такие организации иногда называют «Киллер Б, который умер» (Киллер Б — герой серии японских мультфильмов). *Прим. перев.*

Обычай полагаться на этап стабилизации или повышения надежности, выполняемый в конце проекта, часто обеспечивает плохой результат, потому что проблемы, не найденные и не исправленные в ходе повседневной деятельности, останутся и потенциально могут лавинообразно разрастись в более серьезные последствия. *Прим. авт.*

То, что услуги, сулящие бизнесу самые большие потенциальные выгоды, — браунфилд-системы, не должно удивлять. В конце концов, именно на эти системы полагаются больше всего, они обслуживают больше всего клиентов, или от них зависит получение наибольшего дохода. *Прим. авт.*

Издана: М.: Вильямс, 2006. *Прим. перев.*

Преобразования сверху вниз в стиле «большой шок» возможны, примером служит динамичная трансформация компании PayPal в 2012 г., которую возглавлял ее вице-президент по технологиям, Кирстен Волберг. Однако, как и в случае с любым устойчивым и успешным преобразованием, для этого требуется наивысший уровень поддержки со стороны руководства и постоянная концентрация на движении к необходимым результатам. *Прим. авт.*

Это особенно важно, поскольку мы ограничиваем уровень детализации собираемой информации — ведь время каждого из нас ценно и невосполнимо. *Прим. авт.*

И наоборот, существует много примеров использования инструментов таким образом, который обеспечивает отсутствие изменения поведения. Например, организация переходит к использованию инструмента планирования Agile, но затем настраивает его на процесс каскада, тем самым сохраняя статус-кво. *Прим. авт.*

Сетевой энциклопедический словарь хакерского сленга на английском языке. Содержит более 2300 словарных и обзорных статей по хакерскому сленгу и хакерской культуре. Первым составителем словаря был Рафаэль Финкель. В настоящее время Jargon File поддерживается Эриком Реймондом.  
*Прим. перев.*

Помимо прочего, ORM выполняет абстрагирование базы данных, что позволяет разработчикам выполнять запросы и манипулировать данными, как если бы эти данные были просто еще одним объектом в языке программирования. Среди популярных ORM — Hibernate для Java, SQLAlchemy для Python и ActiveRecord для Ruby on Rails. *Прим. авт.*

Sprouter был одной из многих технологий, использовавшихся в процессах разработки и производства, которые были отброшены компанией Etsy в ходе преобразований. *Прим. авт.*

Однако, как будет пояснено позднее, другие не менее известные компании, такие как Etsy и GitHub, имеют функциональную ориентацию. *Прим. авт.*

Адриан Кокрофт отметил, что «для компаний, которые в настоящее время отказываются от пятилетних контрактов на ИТ-аутсорсинг на внешний подряд, это выглядит, как если бы они были заморожены на эти годы, одни из наиболее прорывных в технологии». Другими словами, ИТ-аутсорсинг — это тактика экономии затрат через обусловленное контрактом помещение в стазис, с твердой фиксированной ценой, предусмотренной графиком ежегодного сокращения расходов. Однако часто это приводит к тому, что организация оказывается не в состоянии реагировать на меняющиеся потребности бизнеса и технологий. *Прим. авт.*

В дальнейшем в этой книге мы будем использовать термин «сервисные команды» как взаимозаменяемый с понятиями «функциональная команда», «продуктовая команда», «команда разработчиков» и «команда доставки». Цель этого заключается в том, чтобы выразить, что команды, занимающиеся разработкой, тестированием и обеспечением безопасности, делают это ради того, чтобы доставлять ценность клиенту. *Прим. авт.*

Как язвительно заметил Джон Лодербах, в настоящее время вице-президент по информационным технологиям в компании Roche Bros. Supermarkets, «каждое новое приложение подобно подаренному щенку. Это не аванс в счет капитальных затрат, когда можно сказать “все, хватит!”. Это неотвратимая необходимость в постоянном обслуживании и поддержке». *Прим. авт.*

Такими же свойствами обладает микросервисная архитектура, построенная на принципах SOA. Один из популярных наборов шаблонов для современных веб-архитектур, создаваемых на основе этих принципов, — «двенадцатифакторное приложение» (12-factor app).

Издана: М.: Вильямс, 2015. *Прим. перев.*

В производственной культуре компании Netflix одно из семи ключевых правил гласит: «Сильно согласованы, слабо связаны». *Прим. авт.*

Черная пятница — пятница после Дня благодарения в США. С нее начинается традиционный рождественский сезон распродаж. *Прим. перев.*

В дальнейшем термины «платформа», «общие сервисы» и «комплекс инструментальных средств» будут в этой книге использоваться как равнозначные. *Прим. авт.*

Эрнест Мюллер отмечал: «В компании Bazaarvoice существовало соглашение, что эти платформенные команды принимают от других команд требования на создание инструментов, но не принимают задания на выполнение работы». *Прим. авт.*

В конце концов, предварительная разработка системы для последующего повторного использования — обычная и весьма дорогостоящая причина отказов во многих корпоративных архитектурах. *Прим. авт.*

Однако, если мы увидим, что весь отдел разработки молча сидит на своих рабочих местах весь день, не общаясь друг с другом, возможно, нам придется найти другой способ вовлечь их в общение, например покупать им обеды, организовать книжный клуб, по очереди проводить презентации типа «обедай и учись», разговаривать с ними, чтобы узнать, какие у кого имеются проблемы, и понять, как мы можем сделать их жизнь лучше. *Прим. авт.*

Scrum — это методология разработки по системе Agile, описываемая как «гибкий, целостный продукт стратегии разработки, при которой команда разработчиков работает как единое целое для достижения общей цели». Она была впервые полностью описана Кеном Швабером и Майком Бидлом в книге *Agile Software Development with Scrum*. В этой книге мы используем термины «разработка Agile» или «итеративная разработка», чтобы охватить различные методы, использующиеся специальными методологиями, такими как Agile и Scrum. *Прим. авт.*

В этом контексте среда определяется как все содержимое стека приложения, за исключением самого приложения, в том числе базы данных, операционные системы, сеть, виртуализация и все связанные с этим настройки. *Прим. авт.*

Большинство разработчиков хотят тестировать свой код, и нередко им приходится очень долго ждать заказанную тестовую среду, чтобы заняться этим. Известно, что разработчики зачастую повторно использовали старые тестовые среды (иногда давностью в несколько лет) или просили кого-то, имеющего репутацию, найти подходящую среду, причем не спрашивая, откуда эта среда взялась, хотя в ней наверняка отсутствовал один или несколько необходимых серверов. *Прим. авт.*

В идеале мы должны находить ошибки до начала интеграционного тестирования, потому что цикл тестирования — это уже слишком поздний момент для организации быстрой обратной связи с разработчиками. Если мы не в состоянии сделать это, то, скорее всего, у нас имеются проблемы с архитектурой, которые необходимо решить. Проектирование систем для тестируемости с целью добавить возможность обнаруживать большинство дефектов с помощью неинтегрированных виртуальных сред на рабочих станциях — ключевой элемент создания архитектуры, поддерживающей быстрый поток и обратную связь. *Прим. авт.*

Первой системой контроля версий была, скорее всего, UPDATE для CDC6600 (1969). Позднее появились SCCS (1972), CMS для VMS (1978), RCS (1982) и так далее. *Прим. авт.*

Можно видеть, что система контроля версий выполняет некоторые из ITIL-конструкций Definitive Media Library (DML) и Configuration Management Database (CMDB), инвентаризируя все необходимое для повторного создания производственной среды. *Прим. авт.*

На следующих шагах мы также внесем в систему контроля версий всю создаваемую нами вспомогательную инфраструктуру, такую как пакеты программ для автоматизированного тестирования и инфраструктура непрерывной интеграции и конвейера развертывания. *Прим. авт.*

Любой, кто выполнял миграцию кода для системы ERP (например, SAP, Oracle Financials и так далее), знаком со следующей ситуацией: когда миграция кода дает сбой, причиной редко бывает ошибка в коде. Намного более вероятно, что миграция не удалась ввиду неких различий в средах, например между средами, использующимися разработчиками и тестировщиками, или тестовой средой и производственной. *Прим. авт.*

В компании Netflix средний возраст копий Netflix AWS составляет 24 дня, при том что 60 % из них имеют возраст менее недели. *Прим. авт.*

Или разрешать его только в чрезвычайных ситуациях, обеспечивая автоматическую отправку копии протокола работы с консолью по электронной почте в отдел эксплуатации. *Прим. авт.*

Весь стек приложений и сред может быть объединен в контейнеры, что может обеспечить небывалые быстроту и скорость работы всего конвейера развертывания. *Прим. авт.*

Термин «интеграция» имеет в DevOps множество немного отличающихся значений. В разработке он обычно означает интеграцию кода, то есть интеграцию нескольких ветвей кода в одно дерево в системе контроля версий. В непрерывной доставке и DevOps интеграционное тестирование означает тестирование приложения в среде, приближенной к производственной, или в интегрированной тестовой среде. *Прим. авт.*

Блэнд описывал, что одним из последствий такого большого числа талантливых разработчиков в компании Google стало появление у них «синдрома самозванца». Этот термин был введен психологами, чтобы неформально описать людей, которые не в состоянии глубоко осознать свои же достижения. В Википедии этот термин описывается так: «Несмотря на внешние доказательства состоятельности, подверженные этому синдрому продолжают считать, что они обманщики и не заслуживают успеха, которого достигли. Успехи они, как правило, объясняют удачей, попаданием в нужное место и время или введением других в заблуждение, будто они умнее и компетентнее, чем есть на самом деле». *Прим. авт.*

Они создали обучающие программы, распространявшиеся через известный информационный бюллетень Testing on the Toilet (который они развешивали в санитарных комнатах), план работы и сертификационную программу Test Certified и провели несколько собраний fix-it («исправь это»), которые помогли командам улучшить автоматическое тестирование процессов, с тем чтобы они смогли воспроизвести потрясающие результаты, которых сумела добиться команда GWS. *Прим. авт.*

В разработке термин «непрерывная интеграция» часто означает непрерывную интеграцию нескольких ветвей кода в общую ветку и проверку того, что интегрированный код успешно проходит модульное тестирование. Однако в контексте непрерывной доставки и DevOps непрерывная интеграция также означает работу в среде, близкой к производственной, и успешное прохождение приемочных и интеграционных тестов. Джез Хамбл и Дэвид Фарли устранили эту неоднозначность, обозначив последнее понятие как CI+. Далее в этой книге термин «непрерывная интеграция» будет всегда использоваться в контексте методов CI+. *Прим. авт.*

Если мы будем создавать контейнеры в конвейере развертывания и использовать такую архитектуру, как микросервисы, то можем предоставить каждому разработчику возможности создания неизменяемых артефактов, когда разработчики собирают и запускают все сервисные компоненты на своих рабочих станциях в среде, идентичной производственной. Это позволяет разработчикам создавать и запускать больше тестов на своей рабочей станции вместо тестирования на серверах и дает нам также быструю обратную связь об их работе. *Прим. авт.*

Мы можем даже потребовать, чтобы эти инструменты запускались до внесения изменений в систему контроля версий (например, выполнялся предфиксационный перехват). Мы также можем запускать эти инструменты у разработчика в интегрированной среде разработки (IDE), в которой разработчик редактирует, компилирует и запускает код, что делает обратную связь еще более быстрой. *Прим. авт.*

Мы также можем использовать в качестве механизма упаковки контейнеры, такие как Docker. Контейнеры обеспечивают возможность «написано однажды, используется везде». Эти контейнеры создаются как часть нашего процесса сборки и могут быть быстро развернуты и запущены в любой среде. Поскольку один и тот же контейнер будет работать в любой среде, мы можем обеспечить согласованность всех наших артефактов сборки. *Прим. авт.*

Именно эта проблема привела к созданию метода непрерывной интеграции. *Прим. авт.*

Существует большая категория архитектурных методов и способов тестирования, используемых, чтобы справиться с проблемами тестирования в случаях, требующих входных данных от внешних точек интеграции, включая «заглушки», «мок-объекты», «виртуализацию служб» и так далее. Это становится еще более важным для приемочного и интеграционного тестирования, в которых необходимо гораздо сильнее полагаться на состояние внешних данных. *Прим. авт.*

Мы должны делать это только тогда, когда наши команды уже оценили автоматизированное тестирование — этим показателем разработчики и менеджеры могут легко манипулировать. *Прим. авт.*

Издано на русском языке: М.: Вильямс, 2011. На обложке русскоязычного издания в качестве авторов указаны Джез Хамбл и Дэвид Фарли. *Прим. перев.*

Начи Нагаппан, Майкл Максимилиан и Лори Уильямс (из компаний Microsoft Research, IBM Almaden Labs и университета Северной Каролины соответственно) провели исследование, которое показало, что команды, использовавшие TDD, выпускали код на 60–90 % качественнее по показателю плотности дефектов по сравнению с командами, не использовавшими TDD, и тратили на это всего на 15–35 % больше времени. *Прим. авт.*

Если процесс отката кода не очень хорошо известен, то потенциальная контрмера — парное программирование отката, с тем чтобы он был лучше документирован. *Прим. авт.*

Это иногда называют антишаблоном water-Scrum-fall. Это обозначает, что организация утверждает, будто использует методы Agile, но в действительности все тестирование и исправление ошибок выполняются в конце проекта. *Прим. авт.*

Ветвление в системах управления версиями использовалось во многих целях, но обычно оно применяется для разделения работы между членами команды по релизам, рабочим заданиям, компонентам, технологическим платформам и так далее. *Прим. авт.*

Включение-выключение выполнения отдельных участков кода — наличие копира, поддерживаемый размер бумаги и тому подобное — осуществлялось с помощью флагов компиляции (`#define` и `#ifdef`).  
*Прим. авт.*

Выпуск продуктов был задержан в связи с (успешным) IPO. *Прим. авт.*

Канареечным тестированием называется метод, при котором развертывание программного обеспечения выполняется на небольшой группе производственных серверов для небольшого числа клиентов, чтобы убедиться, что при работе с реальным трафиком клиентов не происходит ничего страшного.

Клиентская подсистема сайта Facebook была написана преимущественно на PHP. В 2010 г. для увеличения производительности сайта код PHP был преобразован в код C++ с помощью разработанного в компании компилятора HipHop, затем этот код был скомпилирован в исполняемый файл размером 1,5 ГБ. Этот файл затем был скопирован на все производственные серверы с помощью программы BitTorrent, что позволило выполнить операцию копирования за 15 минут. *Прим. авт.*

В ходе своих экспериментов они обнаружили, что команды SOT были успешными независимо от того, кому они подчинялись — отделу разработки или отделу эксплуатации, если команды были укомплектованы правильными людьми и были нацелены на успех SOT. *Прим. авт.*

Выражение из военной терминологии, означает укрепление передовой. *Прим. перев.*

Хорошой метафорой может служить операция «Щит пустыни». Начиная с 7 августа 1990 г. тысячи человек и множество материальных средств в течение четырех месяцев были безопасно развернуты на театре производства, а кульминацией явился единый многодисциплинарный, хорошо скординированный выпуск. *Прим. авт.* («Щит пустыни» — военная операция по освобождению Кувейта. *Прим. перев.*)

Иногда в русскоязычной практике эти термины не переводятся, остаются на английском языке или даются транслитерацией. *Прим. ред.*

Есть и другие пути для реализации сине-зеленого шаблона. Например, настройка нескольких Apache/NGINX веб-серверов на прослушивание на разных физических или виртуальных интерфейсах, использование нескольких виртуальных корневых объектов на серверах с Windows IIS, привязанных к разным портам, с помощью различных каталогов для каждой из версий системы и символьической ссылки, указывающей, которая из сред рабочая (как Capistrano применяется для Ruby on Rails), параллельная работа нескольких версий сервисов или промежуточного ПО, каждая из которых прослушивает отдельные порты, применение двух различных центров обработки данных и коммутация трафика между ними, вместо того чтобы использовать их лишь в качестве горячего или теплого резерва для аварийного восстановления (кстати, использованием обеих сред, как описано выше, мы обеспечиваем и нормальную работу процесса аварийного восстановления) или с помощью различных зон доступности в среде облачных вычислений. *Прим. авт.*

Этот шаблон часто называют «расширение — сжатие», что Тимоти Фитц описывал так: «Мы не изменяем (трансформируем) объекты базы данных, такие как столбцы или таблицы. Вместо этого мы вначале расширяем базу путем добавления новых объектов, а затем, позднее, сжимаем ее путем удаления старых». Кроме того, существует и растет количество технологий, которые предоставляют виртуализацию, систему контроля версий, введение меток и откат баз данных. Это такие, как Redgate, Delphix, DBMaestro и Datical, а также инструменты с открытым исходным кодом, такие как DBDeploy, позволяющие вносить изменения в базы данных значительно безопаснее и быстрее».

Обратите внимание, что канареечные релизы требуют наличия и работы в производстве нескольких версий нашего программного обеспечения одновременно. Однако, поскольку каждая дополнительная версия, работающая в производственной среде, создает дополнительные сложности для управления, мы должны использовать минимальное количество версий. Это может потребовать использования шаблона базы данных «расширение — сжатие», описанного выше. *Прим. авт.*

Cluster immune system была впервые описана Эриком Рисом, работавшим в компании IMVU. Эта функциональность поддерживается также компанией Etsy в ее библиотеке Feature API, а также компанией Netflix. *Прим. авт.*

Один из современных примеров такой службы — программа Gatekeeper компании Facebook. Это сервис собственной разработки, который динамически выбирает, какие функции предоставляются конкретному пользователю, на основе демографической информации, такой как местоположение, тип интернет-браузера и данных профиля пользователей (возраст, пол и так далее). В частности, функция может быть сконфигурирована так, чтобы она была доступна только внутренним сотрудникам, 10 % пользователей или только пользователям в возрасте от 25 до 35 лет. Другие примеры — Etsy Feature API и библиотека Netflix Archaius. *Прим. авт.*

Примерно то же самое писал Чак Росси, технический директор компании Facebook: «Весь код, обеспечивающий функции, которые мы планируем выпустить в течение следующих шести месяцев, уже развернут на наших производственных серверах. Все, что мы должны сделать, — это включить его». *Прим. авт.*

## **101**

К 2015 г. у Facebook было свыше миллиарда активных пользователей, на 17 % больше по сравнению с предыдущим годом. *Прим. авт.*

Эта проблема имеет худшую вычислительную характеристику  $O(n^3)$ . Другими словами, время вычисления растет экспоненциально как функция от количества пользователей, находящихся в сети, размера их списка друзей и частоты изменения состояния «в сети — не в сети». *Прим. авт.*

Издана на русском языке: М.: Вильямс, 2011. *Прим. ред.*

Архитектура компании eBay прошла через следующие фазы: Perl и файлы (версия 1, 1995 год), C++ и Oracle (версия 2, 1997 год), XSL и Java (версия 3, 2002 год), Java на стороне клиента и сервера (версия 4, 2007 год), микросервисы Polyglot (с 2013 года). *Прим. авт.*

Шаблон удушающего приложения предполагает постепенную замену всей системы, обычно старой, другой, совершенно новой. И наоборот, ветвление абстрактным представлением — термин, введенный Полем Хаммантом, — это метод, где мы создаем уровень абстрагирования между областями приложения, которые мы изменяем. Это делает возможным эволюционное проектирование архитектуры приложения, в то же время позволяя любому разработчику закончить работу над основной веткой (мастер-кодом) и отрабатывать непрерывную интеграцию. *Прим. авт.*

MTTR — Mean Time To Repair — среднее время устранения неисправности или неполадки. *Прим. перев.*

Среди примеров можно назвать Sensu, Nagios, Zabbix, Logstash, Splunk, Sumo Logic, Datadog, Riemann. *Прим. перев.*

Буквальный перевод — «билет», здесь — заказ. *Прим. ред.*

Существует большое количество библиотек для логирования, которые облегчают разработчикам задачу по генерированию телеметрии. Стоит выбирать такие средства, которые позволят нам посыпать все логи приложений в централизованную инфраструктуру, созданную на предыдущем шаге. Из популярных примеров можно назвать rrd4j для Java и ruby-cabin для Ruby. *Прим. ред.*

Application performance monitors. *Прим. перев.*

В 2004 г. Жене Ким, Кевин Бер и Джордж Спаффорд описали это явление как симптом отсутствия «культуры причинно-следственных связей», при этом отмечая, что высокоеэффективные организации понимают, что 80 % всех сбоев происходит из-за вносимых изменений и что 80 % MTTR тратится на выявление того, какое же изменение было причиной. *Прим. авт.*

См. application performance monitors. *Прим. перев.* Совершенно другой набор для мониторинга, агрегирования и сбора информации включает в себя Splunk, Zabbix, Sumo Logic, DataDog, а также Nagios, Cacti, Sensu, RRDTool, Netflix Atlas, Riemann и другие. Аналитики часто называют эту обширную категорию инструментов «мониторами производительности приложений». *Прим. авт.*

Создание простой информационной панели должно быть частью создания любого нового продукта или службы: автоматизированные тесты должны подтверждать, что и служба, и панель работают корректно. Это поможет нашим клиентам и упростит развертывание кода. *Прим. авт.*

**114**

В точном соответствии с предписаниями Базы данных управления конфигурациями ITIL. *Прим. авт.*

Особый интерес представляет инструмент Consul, поскольку он создает абстрактный уровень, сильно облегчающий визуализацию архитектуры приложения, мониторинг, работу с блокировками, хранение конфигураций пар ключ — значение, а также объединение хостов в кластеры и обнаружение ошибок. *Прим. авт.*

СТО — Configure-to-Order — главный технический директор (англ.). *Прим. перев.*

Это может быть стоимостью простоя или ценой, связанной с поздним введением новой функциональности. В терминах разработки продукта второй показатель известен как цена промедления (cost of delay). Это ключ к эффективным решениям и приоритизации. *Прим. авт.*



Auto Scaling — Автоматическое масштабирование (англ.). *Прим. перев.*

AWS — Amazon Web Services — Служба облачных веб-сервисов, предоставляемая компанией Amazon. *Прим. перев.*

Сглаживание и другие статистические методики также используются для управления графическими и аудиофайлами. К примеру, сглаживание изображений (или размытие), когда каждый пиксель заменяется средним всех его соседей. *Прим. авт.*

Другие примеры сглаживающих фильтров — взвешенное скользящее среднее или экспоненциальное сглаживание (которые линейно или экспоненциально увеличивают вес более близких к текущей дате наблюдений соответственно) и так далее. *Прим. авт.*

Инструменты для решения проблем такого типа включают в себя Microsoft Excel (который остается одним из самых простых и быстрых способов обработки данных для решения каких-либо единичных вопросов), а также такие статистические пакеты, как SPSS, SAS и проект R с открытым исходным кодом, который в последнее время стал одним из самых широко используемых статистических пакетов. Есть и другие инструменты, в том числе некоторые приложения Etsy с открытыми исходниками, например Oculus, который определяет графики похожей формы, что может свидетельствовать о корреляции, Opsweekly, который отслеживает частоты и количество оповещений, и Skyline, который выискивает аномальные отклонения в системных графиках и графиках приложений. *Прим. авт.*

Благодаря такому подходу вместе с соответствующей архитектурой мы «оптимизируем MTTR, а не MTBF» (Mean Time Between Failures — среднее время безотказной работы. — *Прим. перев.*) — популярный девиз DevOps, описывающий стремление оптимизировать быстрое восстановление работы в противоположность стремлению избежать неудач. *Прим. авт.*

SVP — Senior Vice-President — старший вице-президент (*англ.*). *Прим. перев.*

ITIL определяет гарантию качества так: продукт устойчиво работает в эксплуатации без вмешательства определенный период времени (например, две недели). В идеале такое определение гарантии должно стать частью всеобщего определения понятия «сделано». *Прим. авт.*

Наблюдая работу с кодом на конечных стадиях создания ПО, мы можем открыть новые способы улучшения потока. Так, можно автоматизировать сложные шаги, требующие работы вручную (например, соединять серверные кластеры приложения, которым требуется шесть часов для выполнения задачи), размещать код по пакетам один раз вместо нескольких на разных этапах тестирования и развертывания, взаимодействовать с тестировщиками, чтобы автоматизировать комплексы тестов, осуществляемых вручную (и убрать узкое место для более быстрых темпов развертывания), создавать более полезную документацию вместо того, чтобы расшифровывать записки разработчика по установлению созданного ПО. *Прим. авт.*

Недавно Джейф Суссна попробовал более четко описать, как лучше достичь целей в проектировании пользовательского интерфейса, назвав эту методику «цифровыми разговорами». Задумка этого подхода — помочь организациям осознать путь пользователя при взаимодействии с их продуктом как сложную систему, таким образом расширяя понимание понятия «качество». Среди ключевых идей — проектирование для пользователя, а не для программы, уменьшение закрытости и увеличение обратной связи, проектирование с учетом неизбежности неудач и учеба на ошибках эксплуатации, использование опыта IT-эксплуатации для проектирования и настрой на эмпатию.

*Прим. авт.*

Вероятность того, что ошибки эксплуатации будут быстро устраниться, увеличится, если команды разработчиков останутся неизменными, а не будут расформировываться после окончания проекта.  
*Прим. авт.*

Акт Сарбейнза — Оксли — закон, ужесточающий требования к финансовой отчетности, к процессу ее подготовки, а также к действиям директоров, менеджеров и аудиторов. Введен после ряда крупных скандалов с финансовой отчетностью крупных компаний (таких, как Enron и WorldCom).  
*Прим. перев.*

В организациях с проектным финансированием может не быть разработчиков, которым можно было бы вернуть проект, поскольку команда уже давно распущена или у нее нет денег или времени брать на себя дополнительную ответственность. Возможные меры в таком случае — устраивать периоды блиц-улучшений, на короткое время создавая и финансируя соответствующие команды, либо выводить сервис из эксплуатации. *Прим. авт.*

В этой книге мы используем термин «инженеры IT-эксплуатации», но термин «инженер по обеспечению стабильности сайтов» употребляется как синоним. *Прим. авт.*

В 2016 году Intuit продал Quicken частной инвестиционной компании H.I.G. Capital. *Прим. авт.*

Существует много других способов проводить исследования пользователя перед началом разработки. Среди самых недорогих — проведение опросов, создание прототипов (моделей, созданных с помощью таких инструментов, как Balsamiq, или интерактивных версий с реальным кодом) и тестирование удобства использования. Альберто Савойя, технический директор компании Google, придумал термин «прототип» для обозначения того, что создается до прототипа с целью понять, движемся ли мы в верном направлении. Изучение пользователя настолько дешево и просто по сравнению с разработкой бесполезной функциональности, что приниматься за разработку нового компонента без какого-либо подтверждения его необходимости не стоит. *Прим. авт.*

Контрафактуальное мышление — термин в психологии, описывающий стремление создавать альтернативные версии прошедших событий жизни. В методике обеспечения надежности он используется для описания интерпретаций событий в «воображаемой системе», а не «реальной».

*Прим. авт.*

В этой книге термины анализ кода и анализ изменений будут использоваться как синонимы. *Прим. ред.*

Кстати, скорее всего, список областей высокого риска уже был создан вашим консультативным советом по внесению изменений. *Прим. ред.*

В некоторых организациях парное программирование может быть обязательным компонентом работы, тогда как в других инженеры сами ищут себе напарника для деятельности, требующей особой тщательности (например, перед отправкой готового кода в систему), или для сложных задач. Еще один распространенный подход — установить специальные часы для парной работы, например четыре часа с середины утра до второй половины дня. *Прим. авт.*

Pull request не имеет прямого перевода на русский язык. Термин обозначает процесс улучшения, внесенного кем-то в чужой репозиторий, разработанный ранее, с просьбой к автору оригинала принять эти изменения. *Прим. ред.*

Жене Ким выражает благодарность Шону Дэвенпорту, Джеймсу Фрайману, Уиллу Фарру и Райану Томайко из организации GitHub за обсуждение того, что отличает хороший запрос от плохого.  
*Прим. авт.*

Got Goo? — «Увязли в чем-то?» (англ.). Прим. перев.

Join The Rebellion — «Присоединяйся к восстанию» (англ.). *Прим. перев.*

В январе 2013 г. на конференции re: Invent Джеймс Хэмилтон, вице-президент и заслуженный инженер Amazon Web Services, сказал, что в Восточном регионе США у них было более десяти данных центров, а если учесть, что в типичном данных-центре от 50 000 до 80 000 серверов, то сбой 2011 г. затронул клиентов на более чем полумиллионе серверов. *Прим. авт.*

J2EE — Java 2 Enterprise Edition — набор спецификаций и документации для языка Java, описывающий архитектуру серверной платформы для средних и крупных предприятий. *Прим. перев.*

Такая практика также называется послеаварийным анализом без поиска виноватых (blameless post-incident review) или послеаварийной ретроспективой (post-event retrospective). Стоит отметить схожесть с обычными ретроспективами во многих методиках гибкой разработки или итеративной разработки. *Прим. авт.*

Мы также можем распространить на отчеты анализа ошибок философию Transparent Uptime (блог, в котором описываются и пропагандируются принципы максимальной прозрачности в деятельности ИТ-компаний, автор — Ленни Рачицки. *Прим. перев.*). Вдобавок к публикации показателей сервисов мы можем выкладывать в общий доступ информацию с совещаний по анализу ошибок (возможно, цензурированные). Среди самых популярных общедоступных разборов ошибок — данные, опубликованные командой Google App Engine после значительного сбоя в 2010 г., а также анализ сбоя DynamoDB Amazon в 2015 г. Интересно, что компания Chef публикует в своем блоге заметки с таких совещаний, а также видеозаписи с реальных встреч по разбору причин ошибок. *Прим. авт.*

Генеральный директор. *Прим. ред.*

Издана на русском языке: *Нейгард М.. Release it! Проектирование и дизайн ПО для тех, кому не все равно.* СПб.: Питер, 2016. Прим. перев.

EC2 — Elastic Compute Cloud — веб-сервис Amazon, предоставляющий масштабируемые вычислительные ресурсы в облаке. *Прим. перев.*

Среди конкретных архитектурных решений — жесткое отключение (установка жестких лимитов ожидания ответа, чтобы выход из строя отдельных компонентов не приводил к отказу всей системы), переключатели уровней (проектирование каждого элемента функциональности так, чтобы он переключался на более низкое качество или на менее ресурсоемкий аналог) и удаление возможностей (удаление некритичных тормозящих компонентов с веб-страницы, чтобы они не влияли на впечатления пользователя). Еще одним потрясающим примером адаптивности Netflix, помимо продолжения работы во время сбоя AWS, было то, что инцидент первой степени тяжести был объявлен только спустя шесть часов. Подразумевалось, что рано или поздно сервис AWS будет восстановлен («AWS скоро снова начнет работать... он ведь всегда восстанавливается, да?»). Только спустя шесть часов после сбоя инженеры запустили процедуры по поддержанию непрерывности бизнеса. *Прим. авт.*

Маунтин-Вью — город в штате Калифорния, США, где расположен главный офис компании Google.  
*Прим. перев.*

@hubot «Развернуть owl в производственную среду». *Прим. перев.* OWL — библиотека класса для создания оконных приложений на языках C и Pascal. *Прим. ред.*

Отсылка к выражению Watercooler talk: сотрудники делают небольшие перерывы в работе и идут к куллеру за водой, на ходу и у самого кулера перекидываются парой слов о жизни, о работе и так далее. *Прим. перев.*

Hubot часто выполнял задачи с помощью вызовов скриптов командной оболочки, которые потом можно было выполнить отовсюду, где был доступ к чату, в том числе и с телефона. *Прим. авт.*

Проекты Chrome и Android находятся в другом репозитории, а некоторые алгоритмы, которые держатся в секрете, например PageRank, доступны только избранным командам. *Прим. авт.*

Компилируемым языком в Google был C++, языком сценариев — Python (впоследствии замененный на Go), а языком пользовательского интерфейса — Java и JavaScript с использованием Google Web Toolkit. *Прим. авт.*

В то время в Etsy пользовались PHP, lighttp, Postgres, MongoDB, Scala, CoffeeScript, Python и многими другими языками и средами. *Прим. авт.*

Додзё — зал для тренировок в японских боевых искусствах. *Прим. перев.*

Здесь и далее термин «хакатон» будет использоваться как синоним термина «блиц-обучение», а не в значении «время, когда можно работать над чем угодно». *Прим. авт.*

Кстати, первая конференция DevOpsDays компании Target была проведена по образцу первой конференции DevOpsDays компании ING, организованной Ингрид Алгра, Джэн-Джуист Боумэн, Эвелин ван Льювен и Крим Байтерт в 2013 г. А ее идея, в свою очередь, возникла после того, как сотрудники ING посетили конференцию DevOpsDays 2013 г. в Париже. *Прим. авт.*

## **161**

Фрод (fraud) — вид мошенничества в области информационных технологий, в частности несанкционированные действия и неправомочное пользование ресурсами и услугами в сетях связи.  
*Прим. перев.*

Организация The Open Web Application Security Project (OWASP) — некоммерческая организация, ее цель — улучшение защиты данных программного обеспечения. *Прим. авт.*

Стратегии управления этими рисками включают в себя проведение тренингов для сотрудников, изменение структуры информационных систем, в том числе сетей и программного обеспечения, и создание процессов для предотвращения и обнаружения атак и ответа на них. *Прим. авт.*

Common Vulnerability Scoring System — общая система оценки уязвимых мест. *Прим. перев.*

Среди инструментов обеспечения целостности зависимостей нашего программного обеспечения — OWASP Dependency Check и Nexus Lifecycle. *Прим. авт.*

В качестве примеров инструментов, помогающих тестировать правильность настроек (то есть проверяющих, «как должно быть»), можно назвать автоматизированные системы управления конфигурациями (например, Puppet, Chef, Ansible, Salt), а также такие инструменты, как ServerSpec и «Армия обезьян Netflix» (Netflix Simian Army: Conformity Monkey, Security Monkey и так далее).  
*Прим. авт.*

FISMA — Federal Information Security Management Act — Федеральный закон об управлении информационной безопасностью; FedRAMP — The Federal Risk and Authorization Management Program — Федеральная программа управления рисками и авторизацией; FITARA — Federal IT Acquisition Reform Act — Федеральный закон реформы приобретения информационных технологий.  
*Прим. перев.*

Такие разрешения называются FedRAMP JAB P-ATO. *Прим. авт.*

В ITIL функциональность — это то, «что делает сервис», а гарантии — то, «как сервис предоставляется пользователям и как можно определить, что он пригоден для использования».  
*Прим. авт.*

Чтобы лучше управлять рисками, мы можем также определить специальные правила, например что определенные изменения могут производиться только определенными группами или сотрудниками (к примеру, только инженеры баз данных могут вносить правки в схему базы данных). Традиционно встречи комитетов по изменениям для рассмотрения запросов и составления планов по их воплощению в жизнь проходили раз в неделю. Начиная с версии ITIL под номером 3, изменения можно одобрять удаленно с помощью специального инструмента управления. Там также рекомендуется, чтобы «стандартные изменения определялись рано, при организации процессов управления переменами, чтобы повысить их эффективность. Иначе внесение изменений может привести к высокому уровню бюрократизации и сопротивления этим процессам». *Прим. авт.*

Термин тикет обычно используется для обозначения любой уникально идентифицируемой рабочей задачи. *Прим. авт.*

IPO — Initial Public Offering — первичное размещение акций (*англ.*). *Прим. перев.*

PCI DSS (Payment Card Industry Data Security Standard) — стандарт безопасности данных индустрии платежных карт, разработанный Советом по стандартам безопасности индустрии платежных карт, который учрежден международными платежными системами Visa, Master Card, American Express, JCB и Discover. *Прим. перев.*

«Я мочь иметь токены» — разговорная конструкция, основанная на популярном интернет-меме.  
*Прим. перев.*

Авторы благодарят Билла Мэсси и Джона Оллспоу за то, что они потратили целый день на то, чтобы поговорить с Джином Кимом о своем опыте работы с нормами и требованиями. *Прим. авт.*

CDE — Cardholder Data Environment — сетевая среда, где хранятся и передаются данные платежных карт; часть PCI DSS. *Прим. перев.*

NIST 800-37 — «Руководство по использованию принципов управления рисками в федеральных информационных системах», руководство по трансформации традиционного процесса сертификации и аккредитации в шестиступенчатую систему управления рисками. *Прим. перев.*

Закон Грэмма — Лича — Блайли — закон о финансовой модернизации, принятый в 1999 г., основная цель которого — отмена важных ограничений в финансовой сфере, введенных актом Гласса — Стиголла 16 июня 1933 г.; среди основных положений закона — запрет для коммерческих банков одновременно заниматься кредитными операциями и инвестированием в компании. *Прим. перев.*

Health Insurance Portability and Accountability Act — Закон об отчетности и переносе данных о страховании здоровья граждан, принят в США в 1996 г. *Прим. перев.*

DSDM — Dynamic Systems Development Method, итеративный и инкрементный подход к разработке программного обеспечения, который придает особое значение продолжительному участию в процессе пользователя/потребителя. *Прим. перев.*

## **181**

Stand-up — пятиминутка, подход к проведению совещаний, во время которого все его участники стоят; дискомфорт вынуждает делать такие совещания короткими и насыщенными. *Прим. перев.*

Издана на русском языке: *Poiter M.* Тойота Ката: Лидерство, менеджмент и развитие сотрудников для достижения выдающихся результатов. СПб.: Питер, 2014.

Издана на русском языке: Рис Э. Бизнес с нуля. Метод Lean Startup для быстрого тестирования идей и выбора бизнес-модели. М.: Альпина Паблишер, 2015.

«Независимо от того, что мы обнаружим, мы понимаем искренне верим, что все хотели сделать свою работу как можно лучше, в тех условиях, с теми знаниями, навыками и способностями, ресурсами и информацией, что были на тот момент». *Прим. перев.*

Имеется в виду вышеупомянутая книга указанного автора «Release it! Проектирование и дизайн ПО для тех, кому не все равно». *Прим. ред.*