Павел Рудницкий

SRE

Новый подход к управлению инфраструктурой

Санкт-Петербург
«БХВ-Петербург»
2026

УДК 004.43
ББК 32.973.26-018.1
Р83

Рудницкий П. И.

Р83 SRE. Новый подход к управлению инфраструктурой. — СПб.: БХВ-Петербург, 2026. — 176 с.: ил.

ISBN 978-5-9775-2165-9

Книга служит практическим введением в дисциплину SRE — современный подход к построению цифровых инфраструктур, который обеспечивает практически бесперебойный доступ к сервисам, сайтам и приложениям в условиях меняющихся нагрузок и соответствие заданным SLO (целевым уровням обслуживания). Подход SRE является продолжением и развитием методологии DevOps. В книге рассмотрены приемы сбора и логирования метрик, использование данных телеметрии, профилирование инфраструктуры, выкатывание новых версий системы без приостановки ее работы. Изложенные приемы SRE удобно использовать не только в крупных, но и в небольших компаниях, облегчая и конкретизируя работу сетевых инженеров и упрощая работу сайта, сервиса или веб-приложения для пользователей.

Для системных администраторов, DevOps-специалистов, SRE-инженеров

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Олег Сивченко*
Зав. редакцией	*Людмила Гауль*
Редактор	*Григорий Добин*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн обложки	*Зои Канторович*

Подписано в печать 30.07.26.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 14,19.
Тираж 1300 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-2165-9

© ООО "БХВ", 2026
© Оформление. ООО "БХВ-Петербург", 2026

Оглавление

Предисловие **7**

Глава 1. Принципы SRE **9**
Зачем это нужно? 9
Надежное из ненадежного 11
Как оценить изменения? 14
Автоматизируй это 14
Заключение 16

Глава 2. Прежде всего — коммуникация **17**
Почему ИТ — это в первую очередь про общение 17
Проблема коммуникации 18
Stakeholder'ы внешние и внутренние 19
Как обеспечить прозрачность? 22
Заключение 25

Глава 3. Оцениваем требования **26**
Явные и неявные требования 26
Регуляторные ограничения 29
Оценка изменений 31
А стоит ли это делать? 34
Как говорить с бизнесом на его языке? 35
Заключение 36

Глава 4. Надежность системы в формальных цифрах. SLI/SLO/SLA **37**
Какой бывает надежность? 37
Парадокс надежности — больше не всегда лучше 39
SLO обеспечивает прозрачность 40
Баланс надежности и развития 43
Заключение 44

Глава 5. Что происходит в системе? Мониторинг **45**
Что происходит «под капотом»? 45
Основные компоненты мониторинга 46
Сбор данных. Whitebox и blackbox monitoring 47
Хранение и обработка данных 49
Выбор системы хранения 50

Из чего можно выбрать? 51
Cacti 51
Zabbix 51
Graphite/Carbon 52
InfluxDB 52
Prometheus 52
Последователи Prometheus: Mimir, Cortex, Thanos, VictoriaMetrics 53
Анализ данных. Тренды, корреляции, статистика, поиск аномалий 54
Расчет SLO по метрике *http_requests_total* 55
Более комплексный запрос — подсчитаем Apdex 56
Алертинг 59
Приоритеты аварий 60
Уведомления при нарушениях SLO и Error Budget 61
Заключение 63

Глава 6. Больше информации. Логирование 65
Что дает логирование? 65
Основные компоненты 66
Структурированные и неструктурированные данные 71
Логирование для расследования инцидентов 73
Заключение 75

Глава 7. Еще больше информации. Трассировка, телеметрия и профилирование 76
Какую проблему решает трассировка? 76
Какие проблемы она порождает? 77
Основные решения и компоненты 79
Копаем вглубь: профайлинг 82
Со стороны пользователя: синтетический мониторинг и RUM 85
Заключение 86

Глава 8. Управление инцидентами 87
Что такое инцидент? 89
Основные принципы управления инцидентом 91
Как обеспечить круглосуточный сервис (и не умереть в процессе)? 93
Прозрачность (да, опять) и коммуникация 96
Как подготовится к аварии (и зачем это делать)? 98
Заключение 100

Глава 9. Изменение системы. Управление конфигурацией 101
Зачем управлять конфигурацией? 101
Чем управлять конфигурацией? 103
Terraform / OpenTofu 105
Ansible 105
Pulumi 105
Puppet 106
Chef 106
Cloud-init 106
Netbox 107

С чего начать? ... 107
Как не пожалеть? ... 108
Пара слов о коде ... 111
Собственные инструменты ... 112
Заключение ... 114

Глава 10. Изменение системы. Развертывание новых версий и откат ... 115
Зачем SRE знать о релизах? ... 115
Безболезненный релиз: CI/CD и немного DevOps ... 115
Процесс релиза с точки зрения инфраструктуры ... 117
Подходы к развертыванию: Blue-Green, Canary, Stop&Go ... 118
Blue-Green ... 119
Canary Deployment ... 119
Stop&Go ... 120
Обратимые и необратимые изменения. Откат изменений ... 121
Заключение ... 123

Глава 11. Безопасность ... 124
SRE как способ улучшить безопасность ... 124
Воры ресурсов ... 126
Вымогательство ... 126
Целевая атака ... 126
Анализ способов атак. MITRE ATT&CK и Lockheed Martin KillChain ... 127
Безопасность билда. Подход Shift-Left ... 131
Безопасность развертывания. SBOM, цифровые подписи, принцип наименьших привилегий ... 133
Лирическое отступление: хеширование, асимметричное шифрование, цифровая подпись ... 134
Безопасность машин: mTLS, шифрование, принцип наименьшего доверия ... 136
TLS-сертификаты ... 138
Взаимная аутентификация и mTLS ... 140
Безопасность пользовательского опыта: OAuth/OIDC, SAML ... 141
Надежность инфраструктуры: системы неизменяемых состояний ... 145
Поддержание чистоты: автоматическое обновление (yum-cron, unattended-upgrades, renovate bot) ... 147
Роль SRE в расследовании инцидентов безопасности ... 149
Заключение ... 151

Глава 12. Пользовательский опыт. Как общаться с пользователем и не пожалеть об этом? ... 152
Как всё успевать? ... 152
Frontman-ротация и единая точка входа ... 154
Личная продуктивность и тайм-менеджмент ... 157
Управление нагрузкой и обработка задач в команде. Waterfall и Agile ... 159
Waterfall: старый, но надежный ... 160
Agile: доставка ценностей ... 161
Scrum: тяжелая артиллерия Agile ... 162
Kanban: потоковое управление задачами ... 163

Документация....164
Помогите пользователям решить их проблемы. Self-service и IDP....165
Что такое Guardrails, квоты и политики?....167
Заключение....169

Заключение....171
Благодарности....172

Предметный указатель 173

Предисловие

SRE — это модная тема для обсуждения. DevOps умер, сосланный к разработчикам и ИИ-инструментам, классическое системное администрирование уже никто, кажется, не вспоминает даже. Про SRE много говорят, крупнейшие компании рассказывают, как культура SRE повысила надежность и управляемость сервисов компании и помогает обуздать взрывной рост сложности и удержать надежность под контролем.

Может показаться, что SRE — это что-то из мира больших корпораций уровня Google и Microsoft. У этих компаний инфраструктуры чудовищной сложности, а ресурсы в ИТ кажутся бесконечными. Действия внутри таких инфраструктур влияют на немалую долю Интернета — упавший GitHub или GMail заметит вся планета. Google может себе позволить написать свой веб-сервер, если им не нравится то, что есть на рынке (они это сделали), придумать свой собственный сетевой движок (тоже сделали) и даже свой язык программирования (сделали не один раз).

SRE может показаться концепцией, находящейся на острие управления инфраструктурой, этакая космическая программа сисадминского мира. Специалистам по SRE прекрасно платят, но ждут от них соответствующего уровня подготовки и знаний (Google же, туда после курсов никак). Но на самом деле SRE — это простая и в целом очень прикладная идея (точнее, набор идей). Управление доступностью совсем не требует штата интеллектуалов уровня Брендана Грега или Рассела Кокса. SRE может принести пользу даже в небольшой компании, у которой инфраструктура — это несколько неспециализированных системников, а управляет ими сисадмин-одиночка. SRE опирается на очень простые принципы и решает очень понятные проблемы, одинаковые как для маленьких, так и для больших компаний. Несмотря на разницу в масштабах, пользователи хотят примерно одного и того же, и проблемы концептуально схожи.

Вокруг SRE много мифов и непонимания, и в этой книге я попробую рассказать, какую пользу принципы SRE могут принести относительно небольшой компании. Скажу сразу, эта книга — обзорная. В ней минимум технических деталей и специфики реализации, и это сделано специально. SRE — это инженерный подход, и это концепция, а не конкретный инструмент. Специфика инфраструктуры у каждого своя, и инструменты нужно выбирать под конкретно вашу задачу. В книге описаны именно идеи и концепции, а также даны подсказки, куда можно двигаться дальше, что почитать и попробовать.

Книга рассчитана на читателя с опытом системного администрирования, но не требует серьезных знаний и глубокой подготовки. SRE — это простые, понятные принципы, а как их внедрять и какие брать на вооружение — решать читателю.

ГЛАВА 1



Принципы SRE

Зачем это нужно?

Управление доступностью придумали в Google. Достоянием общественности этот подход стал в 2016 году, когда Google опубликовала первую книгу по этой теме — «Google SRE book» (так же известна как «книга с вараном»). К этому моменту подход и принципы были сформированы, оттестированы и отполированы Google, так что сказать точно, когда именно подход был придуман, — нельзя. Проблема, которую Google решала с помощью *управления доступностью* (оно же инжиниринг доступности, оно же Site Reliability Engineering, SRE), — это проблема роста. Причем как роста ИТ-инфраструктур компании, так и роста профессионалов, которые эту инфраструктуру обслуживают, — сисадминов, девопсов, инфраструктурных инженеров и сетевиков.

Зачем вообще Google это придумала? Проблема в том, откуда вообще берутся инфраструктурные инженеры. Раньше это был чисто ремесленный подход. Представим себе обычную компанию средних размеров. В нее приходит талантливый новичок. Он мало что умеет и еще меньше знает, но обожает свою работу и у него горят глаза. Обычно новичков не ставят управлять инфраструктурой, это слишком большой риск, «удачно» допущенная ошибка вызовет многодневный сбой или вообще потерю данных. Новичка ставят в помощь уже существующему админу. Так новичок становится подмастерьем — он учится у старших товарищей, растет и развивается. Инфраструктура развивается тоже, и спустя какое-то время у хорошего сисадмина (назовем его так, хотя принцип справедлив для любого инженера по инфраструктуре) появляется подсознательное понимание сферы его работы, практически шестое чувство. Хороший инфраструктурщик больше чувствует, а не знает, что витую пару не надо обжимать отверткой (хотя в целом при «прямых руках» так делать можно). Инфраструктурщик мгновенно понимает, что отвалившиеся в бухгалтерии телефоны — это следствие чайника, воткнутого в UPS с коммутатором (скачки напряжения вызывают у управляемых коммутаторов очень странные и трудноуловимые глюки). Хороший инженер, подобно средневековому кузнецу по звуку вентиляторов серверной и мерцанию огоньков коммутационного шкафа понимает, что происходит что-то нехорошее и требующее внимания. А часто даже понимает, что именно и с какой точки надо начинать ремонт. С опытом подмасте-

рье набирается опыта и шишек, которые он собрал в своем путешествии по граблям. Такой специалист — товар штучный, потому что опыт у него уникальный.

Подобный опыт очень плохо формализуется и почти не передается. Причем это даже не вопрос вредности или цеховой солидарности — опыт нельзя подарить, его придется получить самому, сделав такие же ошибки и получив такие же последствия. Попытки сделать знания формализованными, в виде документации, модных какое-то время knowledge base (корпоративные wiki процветают, на внедрении осваиваются приличные деньги) — почти никогда не работают. Сам процесс передачи знаний формализуется очень плохо. В результате в wiki что-то не написали, что-то упомянуть просто забыли, что-то автор документации посчитал очевидным, а где-то его отвлекли, и абзац оборвался на середине. Каждый инфраструктурный инженер становится штучным и уникальным специалистом, неотделимым дополнением инфраструктуры, в которой он работает. Найм такого человека вызывает боль у HR, увольнение превращается в настоящую катастрофу (особенно когда инфраструктура сложная и нагруженная, а штат отдела недоукомплектован и лишних людей нет — т. е. практически всегда), а onboarding новичка — введение его в должность — может легко занять по времени больше испытательного срока.

В Google хлебнули этих проблем полной ложкой. Компания росла очень быстро, времени на полировку и встраивание новых людей «эволюционным путем» у них просто не было. Дополнительной проблемой стала инфраструктура самой Google — она, во-первых, очень большая (шутка ли, 1-е и 2-е места по трафику среди сайтов в мире по версии SimilarWeb). Во-вторых, инфраструктура у Google уникальна (в силу масштаба обычные подходы не работают на таких объемах). То есть опыт обычного инфраструктурного инженера в Google был практически неприменим — нужно было учить специалистов почти с нуля. На что опять же — нет времени, инженеры нужны уже сейчас, а не через пару лет. Чтобы обуздать эту проблему — Google придумали новый подход к управлению инфраструктурой. Принципы довольно простые и выглядят весьма очевидно, когда они уже сформулированы...

- **Инженерия — это измерения**.

  Управление инфраструктурой — это инженерная дисциплина, а не ремесленно-цеховой подход. Инженер отличается от ремесленника в первую очередь тем, что он не чувствует, он знает. Инженер проводит *измерения* и точно знает, в какие моменты поведение системы отличается от оптимального. Понять момент, когда сервис перестал работать, может быть на удивление сложно и неочевидно. Инженер имеет понятные признаки того, является сервис рабочим или нет.

- **Управление риском**.

  В отличие от классического подхода, инженер по доступности не избегает *рисков*. Любое действие в инфраструктуре дестабилизирует ее, а значит, рисков избежать невозможно. Вместо этого инженер по доступности управляет риском — он знает, какая мера риска допустима, и может управлять балансом стабильности системы и выгрузки изменений. Закладывая риски в дизайн своей системы, инженер по доступности предполагает, что абсолютно надежных систем нет

и любой компонент может сломаться. Он готов к такому поведению и имеет сценарий отказа на любой случай.

- **Автоматизация**.

  Инженер по доступности автоматизирует свою работу. Любая неавтоматизированная работа — это потенциальный риск. Кто-то где-то ошибся, кто-то что-то забыл, не записал, не сообщил. На выходе будет непредсказуемое поведение системы. *Автоматизация* — еще и отличный способ передавать свою работу коллегам по команде. Код, в отличие от документации, сам себя объясняет и редко врет.

- **Борьба со сложностью**.

  Инженер по доступности упрощает свою работу. Любое усложнение — это пространство для ошибок, недетерминированного поведения и потенциальных проблем. Любую систему кто-то когда-то будет обслуживать и чинить, позаботьтесь об этих людях заранее. Возможно, это будете вы, в 5 утра воскресенья после масштабного новогоднего застолья. *Сложность* имеет тенденцию к самопроизвольному росту и расползанию в неожиданные места и может очень больно укусить, если не держать ее под контролем.

- **Прозрачность и обратная связь**.

  Инженер по доступности обеспечивает *прозрачность*. Зная, что происходит в его инфраструктуре, он может транслировать это знание «наверх» по цепи управления. Он может управлять емкостью системы и знает, какую нагрузку система выдержит. Это помогает увидеть инфраструктуру с точки зрения экономики (что очень важно для бизнеса). Прозрачность в управлении авариями помогает клиентам — они понимают, что происходит, какие действия предпринимаются и когда инфраструктура вернется к работе.

Надежное из ненадежного

Один из главных принципов SRE — *управление риском*. Риск в инфраструктуре существует всегда, т. к. нет абсолютно надежных систем, ломается всё, даже космические аппараты (казалось бы, эталон надежности) и ядерные реакторы. ИТ-инфраструктура сталкивается с рисками «снизу» (отказы частей инфраструктуры, сервисов-зависимостей) и «сверху» (сбойные релизы, всплески трафика, атаки).

Традиционный подход предлагает рисков избегать: диски отказывают? Возьмем RAID, будем реплицировать данные на два диска сразу, если что — поменяем. Сгорают блоки питания? Нет проблем, возьмем сервер с заменой БП «на горячую», посадим отдельного человека следить за сигналами от сервера и сделаем склад запчастей. Есть риск, что кто-то «поправит» данные в консоли БД командой `DELETE FROM`? Добавим репликацию для базы и бэкап, если что — откатим. Такой подход работает до определенного предела. Но совершенно надежно защититься от сбоев невозможно, и возникают сразу две с половиной проблемы:

- как ни старайтесь, в какой-то момент появится один компонент, который никак нельзя сделать надежным, и вся надежность системы повиснет на ниточке одно-

го критического компонента. Вам очень, очень повезет, если этот компонент не надо будет обслуживать, потому что любое обслуживание — это вывод всего из эксплуатации, а потом ввод обратно;

- в какой-то момент запас по надежности закончится, и инфраструктура «схлопнется». Так как цель была улучшать надежность, а не готовится к провалу, то авария, скорее всего, станет очень неприятным сюрпризом. Очень редко в таком случае есть сценарии отказа и понятный всем алгоритм действий. Еще реже проводятся тренировки, где все участники проверяют, как они должны действовать в случае аварии. Из-за этого в тот момент, когда авария уже произошла, нужно одновременно устранять последствия аварии, восстанавливать рассыпающуюся инфраструктуру и как-то удерживать ситуацию под контролем. Топменеджмент, который в такие моменты висит над ухом, потому что клиенты оборвали все телефоны, не упрощает жизнь совсем. Если на проблемный сервис опирались другие сервисы — вы получите отказ по цепочке, и вам придется восстанавливать не только сбойный сервис, но и всех его соседей;
- традиционный подход к увеличению надежности — это *ригидность* (не будем обновлять, а вдруг сломается!) и *резервирование* (купим диск на случай отказа диска). И то и другое имеет свою цену. Ригидность кончится обнаружением критической уязвимости, когда обновляться придется под риском полностью скомпрометировать систему. В этот момент весь собранный долг по обновлениям ударит разом. Резервирование — это просто дорого, и нужно как-то объяснить руководству, зачем вы покупаете 10 одинаковых дисков, хотя в сервере установлены четыре, и почему нужно потратить условный миллион долларов на железяку, которая даже не используется. И почему нужно «омертвить» миллион вместо того, чтобы вложить эти средства в развитие бизнеса и заработать больше. К такому разговору надо быть готовым и нужно иметь внятные аргументы.

Подход SRE принципиально отличается — вместо того, чтобы избегать рисков, инженер по доступности управляет ими. Риски и нестабильность системы (до определенного предела) закладываются в систему изначально. Да, у нас могут отказать диски на сервере, и что? Мы к этому готовы. Да, может попасться релиз с багами, но мы готовы и к этому. Может упасть связь, доступ к критически важному компоненту, что угодно — у нас есть сценарии реагирования на это, и мы готовы к такой ситуации. При этом риск дозирован — SRE оставляет небольшой «зазор» для рисков, понимая, что риск неизбежен. Такой «небольшой» риск не ставит под угрозу бизнес и репутацию компании. Любые сервисы и любая инфраструктура проектируются с учетом того, что любой сервис в цепочке может перестать работать. Проблему нужно обойти или изолировать — отдать заранее сохраненные данные, показать внятную ошибку, попытаться подключиться снова (retry).

Для управления риском нам нужно понять, какое количество риска сервис может вынести, прежде чем это начнет влиять на пользователей. С одной стороны, мы не хотим влиять на пользователей негативно, потому что они этого не любят. С другой стороны, риск необходим как лекарство — любое изменение порождает риски, а изменения необходимы для развития сервиса и поддержания надежности и безопасности. Процесс поиска итеративный, и сходу нужную цифру найти не получится

наверняка. Для начала процесса улучшений цифру можно взять просто случайную или измерить уже существующую доступность и отталкиваться от нее. В дальнейшем можно итеративно улучшать доступность (а значит, надежность) сервиса. Проблема в том, что улучшение надежности имеет цену — большая надежность означает дополнительную цену оборудования и снижение частоты релизов (каждый релиз — потенциальный риск).

Обычно доступность измеряют в «девятках» — это процент доступности сервиса. Например две девятки означают надежность 99%, а пять девяток — 99,999% доступности. Теперь нам нужна существующая доступность сервиса. Тут есть два варианта:

- доступность через время — мы будем просто измерять время, которое сервис был доступен. Поделив время, которое сервис вообще отвечал на запросы, на всё время работы сервиса — получаем процент времени, когда сервис работал;
- другой способ — выяснить цель существования сервиса. Какое действие сервиса является ключевой целью его существования и без которого сервис не имеет смысла? Имея это действие (или действия), мы можем просто считать каждое такое действие в форме запроса к сервису: сколько пришло и сколько было обслужено. Поделив количество обслуженных запросов на количество запросов всего — мы получаем процент доступности сервиса. Этот подход прост, но в большинстве случаев он (с небольшими поправками) работает хорошо.

В норме текущая доступность сервиса должна быть выше («лучше»), чем цель по доступности. Разница между целью доступности и реальностью — это наш «запас прочности». Мы можем проводить изменения, заниматься обслуживанием, выкатывать релизы агрессивнее. Чем меньше наш запас прочности — тем осторожнее и консервативнее надо себя вести. К моменту, когда запас надежности закончится, мы понимаем, что любое новое изменение дестабилизирует сервис, и потому стараемся минимизировать любые изменения. Подобный «зазор» обычно называется в терминологии SRE «бюджетом ошибок». Разумеется, бюджет ошибок нужно периодически обнулять, но как часто это делать — зависит от инфраструктуры. Обычно период измерения — 30 дней, но сама Google использует квартальный подход. Это не значит, что в конце периода бюджет ошибок обнуляется и наступает «праздник непослушания», когда все крушат инфраструктуру и релизят без оглядки на надежность. Бюджет ошибок нельзя измерять за очень короткий период времени, потому что на таких отрезках он бесполезен (99,9% доступности за сутки — это всего 1,44 минуты простоя). Нам потребуется некий временной период для определения текущего бюджета, чтобы понять не абсолютную цифру, а процент потраченного бюджета. Это дает оценку риска выкатки изменений. При полном бюджете ошибок мы можем рисковать и делать что-то опасное, а если за последние 30 дней мы *сожгли* весь бюджет (традиционный термин для потраченного бюджета ошибок — это «сжигание», не знаю почему) — стоит умерить исследовательский пыл и делать только осторожные задачи по поддержке инфраструктуры.

Как оценить изменения?

Для создания бюджета ошибок мы выбирали целевую доступность, и эта доступность была не абсолютной. Это не логично с точки зрения нормального, классического инфраструктурного инженера: инфраструктура должна быть надежной. Хочется сделать ее как можно более надежной, предусмотрев все возможные проблемы. Казалось бы, зачем закладывать под свою же работу тикающую бомбу планового простоя, почему нельзя просто сделать как можно более надежно? Проблема в том, что всё, включая надежность, имеет свою цену. Подробнее мы поговорим об этом в следующей главе, а пока что небольшой умозрительный эксперимент.

Представим, что у нас есть некий сервис, который приносит, скажем, миллион рублей в год (вообще это мало, просто цифра красивая, считать легко). У сервиса доступность 99,9%, работает он круглосуточно. В году 365 дней (в среднем), в сутках 24 часа, т. е. сервис работает (24 * 365) * 0,999 = 8751,24 часа из 8760. Запланированный простой у нас 8,76 часа (это 8 часов 45 минут), полный рабочий день. Кажется — это много, но попробуем оценить с другой стороны.

Если сервис приносит 1 000 000 рублей за 8751 час (округлим для простоты подсчета), час работы этого сервиса создает 114,27 руб. дохода. Попробуем добавить еще одну девятку: теперь цель по доступности — 99,99% в год. Тогда запланированный простой у нас всего 52 минуты в год, видимое улучшение! Мы выиграли почти 8 часов. А сколько это в деньгах? Возвращаясь к нашей формуле: 114,27 * 8 = 914 рублей. Да, это даже меньше 1000. И это при том, что в нашем эксперименте очень, очень плохая целевая доступность. Этот несложный умозрительный эксперимент хорошо отрезвляет — любое улучшение имеет свою цену и должно иметь смысл.

Улучшение сервиса имеет цену, причем если сервис улучшается линейно, цена растет по экспоненте. Это не значит, что бороться за надежность не нужно. Но надо помнить, что любые изменения не будут «бесплатными», — это дополнительные услуги и оборудование, специалисты, процедуры, и всё это стоит денег.

Автоматизируй это

Культура SRE предполагает упор на автоматизацию. По мнению Google, SRE-специалисты не должны тратить на ручной труд больше 50% времени — иначе у них не останется времени на улучшения. Ручной труд демотивирует — мало кому хочется чувствовать себя приложением к машине и делать одни и те же одинаковые задачи без видимого прогресса. Ручной труд опасен потенциальными ошибками — что-то может быть сделано неправильно, какие-то шаги могут быть перепутаны, что-то окажется забыто. Автоматизация позволяет избежать этого, потому что единожды написанный код выполняется одинаково и дает один и тот же результат. Большая часть ошибок при выполнении кода связана не с тем, что код сломался сам собой, — просто при написании кода какие-то условия не учитывались. Каждый раз, когда такая проблема всплывает, — она исправляется и больше не появляется.

Автоматизация становится надежнее с каждой итерацией. Автоматизация помогает освободить время для действительно важных задач. Кроме того, у автоматизации есть два не совсем очевидных преимущества:

- Во-первых, автоматизация дает возможность проверять и контролировать работу других.

  Внимание человека ограниченно. Можно (и нужно) построить процесс, когда код перед тем, как применяться, будет проверен коллегой. Просмотр и утверждение изменений (код-ревью) пришло в SRE из программирования, но оно существенно снижает вероятность ошибки. Дополнительный свежий взгляд — это лишний шанс на то, что в production не уедет обидная опечатка.

- Во-вторых, автоматизация наблюдаема и повторяема.

  Код документирует сам себя. В больших инсталляциях очень велик риск, что кто-то когда-то настроил сервер, и никто его с тех пор не трогал. Нет документации, нет описаний, никто не знает, что там живет, как настроено и как работает. Попытки починить недокументированную систему «на живую» превращаются в смесь хирургии и криминалистики, когда нужно понять, как система работала, что именно там поломалось и как вернуть ее в рабочее состояние. И всё это одновременно. К сожалению, документация помогает слабо, потому что пишут ее люди. Люди подвержены ошибкам и могут что-то не указать, забыть, потерять или вообще переврать. Плюс документация обычно пишется уже после установки системы, и рано или поздно возникает «дрейф» — документация начинает отставать от реальной инфраструктуры. Кончается всё это тем, что документация теряет связь с реальностью, превращаясь в «вещь в себе», и никак не помогает в работе с реальной инфраструктурой. Попытки выправить ситуацию делают только хуже — команда бросает работу и героическим рывком переписывает документацию (заодно срывая сроки текущих задач). Но поскольку проблема документации концептуальна — дрейф начинается в тот момент, когда документацию закончили писать, и проблема повторяет сама себя.

Изначально в Google предполагали, что site reliability engineer (инженер по обеспечению надежности сайта) — это полноценный разработчик (т. е. программист). Разумеется, это идеальное, а не необходимое условие, повторить такой подход получится, наверное, только в масштабах Google. Инженеры Google пишут огромные системы, и фактически вся инфраструктура Google написана ими, в масштабах Google практически не работают «обычные» общедоступные продукты. Но компании могут брать этот принцип на вооружение, благо обычно не нужно писать совсем всё совсем с нуля. Есть множество инструментов, помогающих упростить работу, автоматизировать тот или иной надоедливый ритуал, заодно сделав его понятным и снизив количество ошибок.

Конечно, автоматизация имеет свою цену (как и любое изменение — см. ранее), автор не предлагает бросить всё и автоматизировать всю инфраструктуру, сведя ее к одной большой зеленой кнопке «сделать хорошо». Автоматизация — это не достижение, а непрерывный процесс, который сложнее начать, чем поддерживать (серьезно, после того как вы опишете существующую конфигурацию в коде, добав-

лять новые части будет намного проще, чем делать это вручную). Автоматизация — это баланс риска отказа и цена усилий создания. Управление доступностью, кроме всего прочего, — это поиск такого баланса. Автоматизировать всё сложно и дорого (и не нужно), а не автоматизировать ничего — слишком большой риск утонуть под своей же сложностью.

Заключение

Может показаться, что подход SRE — это для больших организаций размером с Google — ведь именно в Google этот поход изобрели для решения своих специфических проблем. На самом деле применять SRE-подход можно и в небольших организациях. Причем даже не обязательно брать на вооружение всё — подход предоставляет набор инструментов, и какие использовать — решать вам. SRE выгодно компании — оно помогает создать более надежные и удобные сервисы за счет дозированного управления риском. Это снижает вероятность катастрофы, которую откладывали, а она всё же произошла.

SRE-подход выгоден руководителям, потому что позволяет описать надежность (в частности) и инфраструктуру в целом в цифрах и аргументировать необходимость изменений. Управление требованиями и ожиданиями помогает построить коммуникацию и создать доверие в общении с другими командами.

Рядовым специалистам SRE-подход тоже на руку: измерения показывают состояние системы и снижают вероятность неприятных сюрпризов. Автоматизация уменьшает количество рутинных операций и шанс человеческой ошибки, а подход к измеримой надежности помогает аргументированно спорить с программистами и понять, в каком направлении стоит прилагать усилия.

ГЛАВА 2



Прежде всего — коммуникация

Почему ИТ — это в первую очередь про общение

Инфраструктуры, которые мы строим, существуют не сами по себе, у нас есть клиенты, как внешние (пользователи), так и внутренние. Ситуация, когда в компании инфраструктурой занимается один-единственный человек, тоже очень редка. Когда много людей работают над сложной задачей — *общение* становится критически важным компонентом. Современные системы очень сложны, и почти никогда нельзя увидеть всю систему разом, от начала до конца и от края до края. В любом случае схему придется упрощать, а значит, что-то останется за кадром.

Координация общения и работы — это задача менеджмента, по крайней мере, в теории. Но это не значит, что можно просто игнорировать эту составляющую, повесив всё на своего менеджера, и спокойно сидеть в условном подвале. Хорошо построенное общение помогает узнать о требованиях и проблемах в инфраструктуре до того, как это станет очевидно и начнет влиять на работу (и до того, как проблемы вылезут на административный уровень):

- горизонтальные связи «отдел-отдел» помогут избежать двойной работы (когда два отдела параллельно решают одну задачу) и плохих решений до того, как эти решения дойдут до production и ударят по бизнесу;
- вертикальные связи «начальник-подчиненный» помогут в выяснении требований (все требования, записанные на бумаге, всегда содержат долю неточности, а любую архитектурную проблему проще решать как можно раньше). Кроме того, вертикальная связь поможет узнать курс компании, понять, к чему стоит готовиться, и заложить изменения заранее.

В большинстве компаний инфраструктура воспринимается не как источник дохода, а как затраты (доходы приносят продажи, а ИТ только поддерживает процесс путешествия денег из кошелька клиента на счет компании). Это ставит инфраструктуру в положение «гадкого утенка» — первого кандидата на сокращение расходов и последнего на их увеличение. ИТ-инфраструктура вносит риски для бизнеса, т. к. «удачно» упавший сервис может лишить компанию серьезной доли годовых доходов или даже подвести руководство под тюремные сроки. Построение

прозрачности — одна из важнейших задач SRE, клиенты и бизнес должны понимать, как работает система, которой они доверяют. Прозрачность создает доверие, потому что понятно, чего ожидать от инфраструктуры в частности и от всего «большого ИТ» в целом.

Сложность современных ИТ-систем повышает требования к общению еще больше — в больших компаниях параллельно работают десятки, сотни, а иногда и тысячи компонентов. Все эти компоненты могут зависеть друг от друга порой неочевидным образом. Каждый компонент разрабатывается отдельной командой, и надо как-то «подружить» их между собой. С административной точки зрения это делается системными архитекторами, которые спускают сверху ADR (решения архитектурного совета), но все такие решения имеют определенную долю абстракции (если делать их конкретными — работа компании будет парализована бюрократией, потому что архитекторам придется принимать абсолютно все решения). При этом инфраструктура становится местом, где все эти компоненты наконец встречаются, и SRE должно стать тем ключевым звеном, которое координирует связи компонентов между собой.

Проблема коммуникации

В теории общение — это очень просто. Как SRE-специалистам нам нужно:

- ❒ обеспечить прозрачность ожиданий от инфраструктуры.

  Инфраструктура надежна и доступна только до определенного предела. Надо обозначить этот предел и показать, что произойдет, когда он будет достигнут;

- ❒ обеспечить разработчикам быструю обратную связь от инфраструктуры.

  Пришел сбойный релиз? Разработчики должны узнать об этом до того, как пользователи придут штурмовать офис, или про вас напишут в газетах. Релиз ведет себя не так, как ожидалось? Аналогично — быстрая обратная связь поможет разработчикам принимать осознанные решения (заказ дополнительного оборудования, откат релиза, выкатка исправлений);

- ❒ получить обратную связь от разработчиков.

  Тесты падают в половине случаев, и разработчики час ждут пайплайн? Об этом стоит узнать заранее. Разработчики вынуждены править базу руками, потому что у них нет инструментов для автоматизации? Это тоже пространство для улучшения. Проблема в том, что разработчики могут иметь выученную беспомощность — они могут даже не знать, что проблему можно починить, и вынуждены искать обходные пути;

- ❒ обеспечить прозрачность для пользователей.

  Мы сломались? Пусть пользователи узнают об этом. Когда мы сломались, какой статус (мы изучаем проблему, чиним, вернулись и наблюдаем). Пользователь должен иметь обратную связь и понимать, что происходит, даже если происходит что-то не то. Нет ничего хуже, чем сказать клиенту, что оплата прошла, и молча потерять деньги;

- обеспечить прозрачность для бизнеса.

  Бизнес принимает решения о деньгах, и ему нужны максимально точные данные. Сколько стоит инфраструктура? День, час работы? Сколько пользователей мы обслуживаем прямо сейчас? Сколько можем обслужить? Сколько будет стоить увеличение нагрузки в два-десять-сто раз? Сколько денег тратит каждая команда? Есть авария — сколько времени займет устранение? Что затронуто? У бизнеса могут быть внешние обязательства, о которых они знают, а вы нет. Например, в контракте могут быть прописаны штрафы за недоступность сервиса. Точная и открытая информация позволит принимать информированные решения и правильно выбирать приоритеты.

Конечно, в теории теория и практика одинаковы, но на практике это не так. Команды изолированы друг от друга и часто не имеют контекста о работе соседей. Менеджмент естественным образом изолирован от подчиненных, и если менеджмент с определенной долей достоверности получает информацию «снизу» (через отчеты), то обратная связь «вниз» не доходит почти никогда. В редчайших случаях компании проводят глобальные встречи (таун-холлы), когда менеджмент рассказывает, как они вообще видят ситуацию и чего вообще хотят. SRE-команда предоставляет сервис (инфраструктуру) всем остальным командам, и это кратно усложняет всем жизнь — команды не понимают, чем занята инфраструктура, почему всё так плохо, медленно и ненадежно и опять сломалось, а задачи живут неведомой жизнью и непонятно, когда за них кто-либо будет браться. И будет ли кто-то их делать вообще.

К счастью, есть несложные шаги, которыми можно поправить ситуацию. Сразу скажу, что коммуникация — это процесс, а не достижение. Нельзя «внедрить коммуникацию» и решить, что теперь всё стало хорошо, — процесс надо поддерживать. К счастью, коллеги быстро обнаруживают удобство работающей коммуникации и понимают, что поддерживать ее проще, чем начинать

Stakeholder'ы внешние и внутренние

Мы работаем не на себя — у нас всегда есть заказчик. Вообще, инфраструктурный инженер, в какой-то мере всегда обречен работать на кого-то, потому что инфраструктура не имеет ценности сама по себе — ценность приносит сервис, который она предоставляет: надежный, быстрый и безопасный способ запускать приложения, которые для клиента пишут программисты. Разработчики — это наши клиенты, бизнес — клиент разработчиков, а клиенты (те, кто платят бизнесу деньги) — это клиенты бизнеса. Заказчик (другой термин — stakeholder) — это не обязательно человек, который дает инфраструктурной команде задачу, но именно stakeholder создает цель нашей работы. Stakeholder — это тот, у кого есть требования, и тот, кто несет в конечном итоге ответственность за реализацию (или ошибки) в этих требованиях. Мы выполняем работу, которая соответствует требованиям stakeholder'а, а stakeholder обеспечивает обратную связь (получил ли он то, чего хотел).

Вообще, очень полезно знать, как именно работает бизнес, и кто в конечном итоге платит нам зарплату. Большинство из нас, спецов по инфраструктуре, работает

в коммерческих компаниях, т. е. у нас есть внешний заказчик, который нам платит (это может быть один большой клиент или много маленьких). Даже если ваш работодатель — это некоммерческая организация или государственное учреждение (например, налоговая служба, суд или ЗАГС) — заказчик есть всегда. Понимание того, откуда бизнес берет деньги, очень помогает в планировании и работе:

- у разных клиентов есть выраженные паттерны поведения. Например, самый важный день для компаний, торгующих цветами, — это 7 марта. В этот момент цветы сто́ят как золотые слитки, а недостачу роз на продажу могут покрыть фрахтом самолета из Аргентины;
- понимание того, откуда берутся деньги, помогает в приоритетах. Если одновременно сломалось несколько вещей — придется выбрать, что чинить первым (про приоритеты починки мы поговорим позже). Зная, что приносит деньги бизнесу, будет проще понять, за что хвататься в первую очередь.

Как узнать, кто наши клиенты? Проще всего поговорить с бизнесом. Обычно до менеджмента легко достучаться, искренний интерес и видимое желание понять и улучшить ситуацию открывают множество дверей. В маленьких компаниях можно пообщаться с генеральным директором напрямую, в компаниях побольше вопросы о том, откуда бизнес растет, можно задать отделу продаж или менеджеру по развитию бизнеса (BDM). Не нужно спрашивать про все детали (всё равно не расскажут), но нужно понять некоторые фундаментальные моменты.

- Во-первых, в чем состоит бизнес компании? Мы производим и продаем? Покупаем, а потом продаем? Обслуживаем (оказываем какие-то услуги)? Обычно на этом этапе могут всплыть ограничения, важные для инфраструктуры, например требования закона по работе с оборудованием и данными. Этим особенно славятся компании, работающие с финансовыми, медицинскими персональными или государственными данными. Попытка обойти или нарушить требование регулятора может окончиться проблемами, вплоть до уголовных.
- Во-вторых, будет здорово получить портрет типичного клиента. Кто он? Это большой бизнес (т. е. у компании один-два-три крупных ключевых заказчика) или мы обслуживаем миллионы маленьких частных клиентов — бизнес в стиле McDonalds или IKEA, где огромный поток посетителей и нужно каждому предоставить стабильный предсказуемый сервис. Какое у этого клиента поведение? Есть ли выраженные паттерны поведения? Например, в бюджетных организациях горячий период — конец финансового года. Это связано с тем, что бюджет выделяется ровно на год и нужно освоить его к концу года. В течение года организации экономят (выйдешь за рамки бюджета — получишь взыскание), а в конце года лихорадочно осваивают наэкономленное (если не потратить весь бюджет — в следующем году могут дать меньше). Из-за этого возникают странноватые ситуации, когда автопарк за три дня до нового года покупает партию автобусов (потому что деньги есть), а ведь автобусы — техника штучная и делаются обычно на заказ. Резко активизируется под новый год и торговля подарочными вещами (а значит, никаких релизов с середины ноября из-за риска сломать сервис), а стриминговые сервисы получают всплеск нагрузки при каждом крупном медиа- или спортивном событии.

- Третье полезное (но не обязательное), что мы можем узнать, — это стадия развития. Для нас, как для инфраструктурных инженеров, значение имеют три первых этапа: стартап, скейлап и зрелая организация:
  - *стартап* активно ищет своего клиента и бизнес-модель. В любой момент всё может поменяться. Работать в такой компании очень интересно, но может выматывать постоянное состояние гонки, потому что стартап бежит, пытаясь обогнать собственную тень. Стартап прощает некоторую неряшливость в процессах и коде, потому что большая часть того, что делается в стартапе, — пойдет на выброс через год. За свой личный опыт работы в стартапах я ни разу не видел ни одного технического решения, которое прожило бы больше года. При этом стартап не означает, что можно проигнорировать качество и делать тяп-ляп, — нагрузка может взлететь, и маленькая ошибочка больно укусит в самый неожиданный момент. Стартап предполагает максимально универсальные решения (пусть и ценой высокой стоимости и меньшего удобства), а также закупку сервисов вместо их построения (так быстрее, а скорость для стартапа — это всё);
  - *скейлап* — это момент, когда стартап нашел свою нишу на рынке и теперь стремительно растет. Уютная компания из пяти человек и шефа в свитере, который приносит кодерам кофе и пиццу, превращается в стеклянный офис, появляется секретарь и начальники отделов. Скейлап — это хороший момент начать строить процессы и считать деньги. Неоптимальные решения не опасны на стадии стартапа, когда стартап активно ищет свою долю рынка и инвестор готов рисковать (и покроет ошибку деньгами в определенных пределах, рабочая бизнес-модель важнее). Скейлап — это ровно тот момент, когда надо начинать считать, иначе можно получить счет, который очень неприятно удивит. Решения, заложенные на этом этапе, станут основой информационной инфраструктуры компании, изменить архитектурное решение в уже устоявшейся системе будет очень сложно (и, как следствие, дорого). Скейлап стремительно наращивает всё: клиентов, денежный оборот, численность коллег. Инфраструктура должна быть готова к росту. Так же это момент, когда стоит задуматься о документации, потому что уже невозможно удержать всё в голове. Человек, чья голова хранит хотя бы часть инфраструктуры, превращается в блокирующий фактор, потому что этот человек — единственный, кто может чинить и безопасно менять кусок, за который он отвечает;
  - *зрелая компания* — это компания с именем. Устоявшийся бизнес, занимающий стабильную нишу рынка (без сильного роста или снижения), с устоявшимися процессами. Такой бизнес обычно слегка ригиден (особенно если рынок консервативен), имеет развитые процессы и архитектурные решения. Менять такие решения сложно, и делать это нужно очень аккуратно, чтобы ничего случайно не поломать. Зрелые компании обычно страдают от бюрократии, самоизоляции разработчиков (каждый отдел живет в собственном маленьком мире) и устаревшей документации (потому что инфраструктура живет своей жизнью и обновлять документацию забывают). Зрелые компа-

нии — это мир оптимизаций, улучшения горизонтальных связей и битв за прозрачность и метрики.

Разные стадии развития требуют концентрации на разных моментах. Если стартап ищет, куда ему расти, и требует скорости и гибкости, то скейлап учит архитектуре и построению процессов. Зрелая компания требует в первую очередь аккуратности и умения балансировать интересы среди большого количества конкурирующих между собой акторов. Отделы и начальники тянут одеяло в разные стороны, и сломать работающий процесс может оказаться просто, а чинить его потом будет сложно.

Кроме клиента, который является нашим внешним заказчиком, есть заказчик внутренний. Это разработчики, которые пишут (и запускают внутри инфраструктуры) разное ПО, это могут быть внутренние клиенты (если инфраструктура обеспечивает бэк-офис — например, почту или телефонию). Я очень рекомендую собрать где-то в одном месте список всех команд с описанием, кто над чем работает. Это может быть wiki-база (например mediawiki, Atlassian Confluence, Docusaurus, Gitlab WikiPage). Вообще, база знаний, даже самая простая — очень полезный инструмент для инфраструктурных инженеров. Даже самая неструктурированная информация, которая где-то записана, намного лучше самой лучшей памяти.

Знать заказчика важно, потому что он помогает нам с приоритетами и обратной связью. Заказчик внешний (клиент) делает это в лице менеджмента. Менеджмент знает (должен, это их работа), как взаимодействовать с клиентом и чего клиент хочет. У менеджеров можно узнать, что клиенту нужно и когда. Пример про цветы 7 марта и предновогодние распродажи как раз отсюда — менеджеры планируют такие вещи заранее. Синхронизация с менеджментом поможет подготовить инфраструктуру и не получить неприятный сюрприз (или, что еще хуже, устроить такой сюрприз клиентам).

Внутренний заказчик — это команды разработки (или «соседи» по инфраструктуре — например, техподдержка офиса). Они дают бесценную обратную связь о том, как им работается с инфраструктурой и чего им не хватает. Запросы могут быть неожиданные и странные, но для разработчиков это может быть критически важно или способно сильно облегчить им работу и жизнь. Счастливый разработчик — это продуктивный разработчик, он пишет быстро и катит изменения без ошибок. Это делает счастливым клиента, а значит, в конечном итоге всех нас.

Как обеспечить прозрачность?

Прозрачность ожиданий (как от инфраструктуры, так и от самих инженеров) — краеугольный камень движения SRE. Чтобы обеспечить прозрачность, нужно сначала понять, как она выглядит:

- клиент (в широком смысле) хочет быть услышанным. У stakeholder'а должна быть возможность связаться с инфраструктурной командой, если что-то пошло не так или если чего-то хочется;
- если инфраструктурную команду о чем-то попросили и было дано какое-то обещание, то заказчик должен знать о том, когда его просьба будет выполняться,

и на какой стадии находится выполнение. Нет ничего хуже для построения доверия, чем пообещать что-то сделать и забыть. Положить задачу в список, где ясно видно, что ее не делают прямо сейчас (потому что делают другие задачи), намного лучше. Никто не обидится, если приятное дополнение сделают через полгода из-за того, что инфраструктурный отдел перегружен и чинит аварии. Задачи в бэклоге показывают, что никто не забыт, а их приоритет демонстрирует, когда задачу будут делать (и почему не делают);

- инфраструктура должна иметь обратную связь так же, как и команда. Если разработчику нельзя что-то делать — он должен понимать причину. Если действия разработчиков привели к проблемам — они должны узнать об этом. Очень важно в этом месте не обвинять разработчиков в том, что они такие-сякие выкатили ломаный релиз и всё сломали. Ошибки нормальны в инжиниринге инфраструктуры, но на этих ошибках надо учиться. А чтобы на них учиться — их надо знать.

Чтобы услышать заказчика (опять же в широком смысле) — нужно предоставить ему какой-то простой *канал связи*. Это должен быть максимально простой, понятный и желательно асинхронный канал связи, доступный всем в компании. Почему простой? Потому что если сделать способ связи сложным — заказчики начнут его «обходить» и будут, например, звонить. Или вообще приходить в гости ногами. Асинхронность важна, чтобы управлять отвлечениями. Если канал связи синхронный — придется посадить кого-то на телефон, буквально работать диспетчером и отвечать на звонки. Обычно в компании есть корпоративный мессенджер (slack, MS Teams, Jabber, Telegram). Для простой и быстрой связи можно создать в мессенджере группу (групповой чат), где сидят инфраструктурные инженеры и куда любой желающий может написать. Обязательно нужно выделить специального человека (фронтмена), который будет этот чат периодически просматривать и реагировать на входящие сообщения. Иначе заказчик посчитает, что его игнорируют и... Правильно, будет приходить ногами. Мой опыт показывает, что лучше создать не один чат, а три — чтобы избежать замусоривания (но это ситуационно — в вашем случае может работать один, а может, вам потребуются 5 или даже 10):

- общий чат инфраструктурной команды.

  Все вопросы к инфраструктурной команде — сюда. Именно этот чат нужно постоянно мониторить. Если у нас есть выделенный «дежурный по запросам» — это разгружает остальных инженеров, потому что ясно, кто должен отвечать. При этом запросы пользователей не остаются без ответа. Кстати, для этой должности лучше иметь ротацию, чтобы дежурный не погряз в рутине и не выгорел в угли;

- канал с уведомлениями о крупных инфраструктурных изменениях, которые могут затронуть остальных.

  Это достаточно важные анонсы, так что их лучше не писать в общий чат (в зависимости от численности компании и ее культуры в общем чате могут быть десятки сообщений в день — анонс потеряется, «пришпиливать» его бесполезно, пины никто не читает). Анонсы — первое место, куда пойдут смотреть, если

что-то сломалось. На этот канал хорошо бы подписать лидов всех команд разработки, чтобы они были в курсе происходящего;

- ❒ чат аварий.

  В случае инцидента нужно какое-то место, где станет координироваться починка. Отдельный выделенный чат будет работать лучше, чем общий, — вся беседа сразу перед глазами, можно создать митинг сходу (подробнее об управлении инцидентом мы поговорим позже). В этом чате точно нужны все лиды всех разработческих команд, все менеджеры (на случай эскалации) и кто-то, кто может взаимодействовать с внешним клиентом (техподдержка, клиентский менеджмент) — на случай, если придется делать публичный анонс. Этот же чат может помочь в составлении хронологии аварии.

Следующий этап улучшения прозрачности — *учет задач*. Обычно в компании уже есть тикет-система, но если ее нет — самое время начать ее использовать, благо их сотни, от мощных облачных и коммерческих (ужасная Atlassian Jira, прекрасный JetBrains YouTrack), так и бесплатных (RedMine, Vikunja, древний как динозавр Mantis). Любая задача, которая требует больше 5–10 минут времени дежурного (см. раздел про чат), — должна приехать в тикетную. Есть разные варианты работы с тикетами (как минимум Scrum и Kanban), главное, чтобы тикеты были в одном месте (где заказчик может тоже на них посмотреть), у тикетов был четко видимый статус: тикет ожидает, в работе или готов. Даже самый-самый простой учет задач намного лучше, чем никакой. Главное — все задачи должны быть в одном месте. Если где-то что-то обсуждалось — любое обещание должно быть зафиксировано.

В тикетах обязательно должно быть отражено следующее:

- ❒ как уже упоминалось — все задачи должны быть записаны, и они должны быть записаны в одном месте. Чем больше у вас мест, где вы храните списки задач, — тем выше вероятность того, что что-то потеряется;
- ❒ у задачи должен быть статус и workflow — путь, по которому задача «путешествует» в процессе выполнения. Самый примитивный flow состоит из трех статусов: «ожидание», «в работе» и «готово». В зависимости от ситуации и конкретных требований flow может меняться — может иметь больше статусов, пути движения задач могут также меняться;
- ❒ у задачи должны быть заказчик (stakeholder, для кого она делается) и владелец (тот, кто делает задачу). Одномоментно у задачи может быть только один stakeholder и только один владелец. Коллективная ответственность — это ничья ответственность: каждый подумает на кого-то другого, и задача «повиснет в воздухе»;
- ❒ у заказчиков должна быть возможность видеть загрузку команды — сколько всего задач, какие задачи сейчас выполняются. Это позволит избежать неприятного вопроса «почему задача не делается». Статистика задач может стать хорошим аргументом к расширению отдела (если команда очевидно зашивается под грузом задач) или указывать на проблемы в определенной области (если инфраструктура чинит финансовую базу каждую неделю на протяжении года — в ней явно есть проблема).

Получить обратную связь от команд будет несколько сложнее, потому что команды обычно воспринимают инфраструктуру как данность. Даже если она работает плохо — обычно это считается неотъемлемым свойством конкретной компании, как закат солнца в конце дня. Мало кто критикует инфраструктуру, которая работает давно, даже если она работает плохо, — это просто считается неприятной нормой, и разработчики, люди умные, часто находят весьма креативные способы обойти проблемы, с которыми они вынуждены жить. Команды могут бурно среагировать только на резкое изменение (всё сломалось или, наоборот, внезапно хорошо заработало).

Цель управления доступностью — непрерывное совершенствование, в том числе совершенствование инфраструктуры. Чтобы получить информацию от команд, могут помочь регулярные периодические митинги-встречи (их не нужно проводить часто, обычно хватает ежемесячных), где командам предоставляется возможность поделиться своими сложностями и пожеланиями. Главная проблема таких встреч — не скатиться во взаимные обвинения, так что вам потребуется морально устойчивый модератор, который сможет отделить эмоции от рациональных требований и пожеланий. На встречу стоит пригласить лида команды обязательно и остальных участников по желанию. Все требования, пожелания и вопросы (какими бы странными они ни казались) нужно обязательно записывать. В идеале после каждого митинга можно создавать маленький отчет и класть его в ту же базу знаний. Каждый следующий митинг можно начинать с ретроспективы предыдущего, чтобы было понятно, есть ли улучшения. Заодно это покажет команде, что инфраструктурная команда прислушивается к запросам разработчиков и ценит их время. На всякий случай напомню, что если запрос команды предполагает какую-то работу (а не просто «поговорить с менеджерами, чтобы не самая важная команда не уходила в отпуск в полном составе») — нужно создавать на такие задачи тикеты и класть их в свою систему учета задач. Это не значит, что сразу после встречи надо бросать всё и бежать выполнять запрос команды — такой подход разрушит фокус команды на уже запланированной работе. Но все задачи должны быть записаны и сложены в одно место.

Заключение

Коммуникация очень важна для построения надежной инфраструктуры. Другие команды и заказчики «сверху» будут в курсе, чем занята команда и какой у нее предел нагрузки. С другой стороны, инфраструктурная команда своевременно получит информацию о планах (что позволит подготовится до того, как небо упадет на землю). Связь с командой разработки поможет обменяться опытом и раскроет слабые места инфраструктуры, не всегда очевидные для тех, кто эту инфраструктуру построил (т. к. далеко не всегда инфраструктурой пользуется тот, кто ее создавал). Построение доверия между командами — дело, требующее времени, но прозрачность работы, открытость и управление ожиданиями stakeholder'ов очень помогают.