ГЛАВА 1

Четыре ключевые метрики в действии

Эндрю Хармел-Лоу

Нет ничего зазорного в том, чтобы считать, что новаторская книга «Accelerate»[1] (изд-во IT Revolution Press, 2018) доктора Николь Форсгрен (Nicole Forsgren), Джеза Хамбла (Jez Humble) и Джина Кима (Gene Kim) — это исчерпывающее руководство по повышению производительности доставки ПО, и эта производительность измеряется четырьмя простыми, но емкими ключевыми метриками.

Кроме того, моя собственная работа над трансформацией во многом основана на рекомендациях из этой книги, и я полностью согласен с ее содержанием. Однако же, на мой взгляд, книга не отменяет потребность в более глубоком изучении, а, наоборот, должна стать предметом дальнейшего обсуждения и анализа. Это позволит обмениваться опытом и объединить сообщество архитекторов, стремящихся к улучшениям. Я надеюсь, что эта глава будет способствовать развитию дискуссии.

Я убедился, что если команда использует четыре ключевые метрики — частоту развертываний (deployment frequency), время выполнения доставки (lead time for changes), долю неуспешных изменений (change failure rate) и время восстановления сервиса (time to restore service)[2] —

---

[1] Форсгрен Н., Хамбл Д., Ким Дж. «Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации».

[2] Перевод дается по книге «Ускоряйся». В исследовании состояния DevOps в России (см. https://express42.com/state-of-devops/) эти метрики называют частотой развертываний, сроками поставки, неуспешными изменениями и временем восстановления соответственно. — *Примеч. науч. ред.*

описанным далее в главе способом, то это помогает ей обучаться и понять, насколько необходима высококачественная, слабосвязанная и модульная архитектура, которую просто развертывать, тестировать, наблюдать и поддерживать. Грамотное применение этих метрик позволяет архитектору ослабить контроль. Вместо диктата и жесткого управления вы можете использовать четыре ключевые метрики, чтобы вовлекать команду в диалог и стимулировать ее улучшать программную архитектуру как общее, а не личное дело. Такой подход позволяет постепенно продвигаться к связной, целостной, модульной и отказоустойчивой архитектуре, которая нативна для облачных сред и которую проще тестировать, запускать и контролировать.

В следующих разделах я покажу, как запустить эти четыре ключевые метрики, и, что важнее, как вам и вашим командам разработчиков эффективнее всего использовать их, чтобы сосредоточиться на непрерывном улучшении процессов и отслеживании прогресса. Основное внимание я уделю практическим вопросам:

- визуализации схемы с четырьмя ключевыми метриками;
- сбору трех необходимых исходных точек данных;
- расчету и отображению самих метрик.

Кроме того, я расскажу о преимуществах архитектуры, работающей на продакшен.

Определение и сбор данных

> *Парадигмы — это источники систем. Из них, из общих соглашений внутри общества о природе реальности, возникают цели системы, потоки информации, обратные связи, запасы, потоки и все остальное, что касается систем.*
>
> *Донелла Медоуз (Donella Meadows), Thinking in Systems: A Primer*[1]

Четыре ключевые метрики вытекают из схемы, лежащей в основе упомянутой выше книги «Accelerate». Я начинаю с этого утверждения, потому что при чтении этой главы базовую схему важно держать в голове. В своей простейшей форме модель представляет собой пайплайн (или «поток»)

[1] Медоуз Д. «Системное мышление. Как создавать и улучшать системы в бизнесе и жизни».

действий, который запускается, когда разработчик отправляет изменения кода в систему контроля версий, и завершается, когда эти изменения поглощаются работающей системой, доставляя работающий сервис пользователям. Эта схема представлена на рис. 1.1.

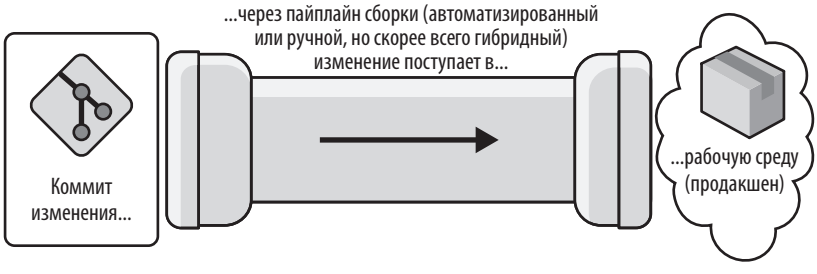

Рис. 1.1. Базовая схема, лежащая в основе четырех ключевых метрик

Для ясности визуализируем, что именно измеряют четыре ключевые метрики в рамках этой модели:

Частота развертываний (Deployment frequency)

Количество отдельных изменений, которые выходят из пайплайна за единицу времени. Эти изменения могут представлять собой «единицы развертывания»: код, конфигурацию или все вместе, в том числе, например, новый функционал или исправление дефекта.

Время выполнения доставки (Lead time for changes)

Время, которое требуется для того, чтобы завершенные разработчиком изменения в коде или конфигурации прошли через весь пайплайн.

В сочетании эта первая пара метрик позволяет измерять *пропускную способность разработки* (development throughput). Ее не следует путать с циклом разработки по методологии Lean или общим временем выполнения (lead time), включающим время на написание кода, отсчет которого иногда начинается с момента, когда продакт-менеджер впервые придумывает идею для новой фичи.

Доля неуспешных изменений (Change failure rate)

Доля изменений, выходящих из пайплайна, которые вызывают сбой в работающем сервисе. (Конкретные критерии, определяющие «сбой», будут рассмотрены далее. Пока что просто считайте сбоем все, что мешает пользователям вашего сервиса выполнять свои задачи.)

Время восстановления сервиса (Time to restore service)

> Время, которое требуется после того, как сервис столкнулся со сбоем, чтобы осознать проблему и доставить исправление, которое восстановит работу сервиса для пользователей[1].

Вместе эта вторая пара метрик описывает *стабильность сервиса*.

Сила этих четырех ключевых метрик — в их сочетании. Если вы повышаете пропускную способность разработки, но при этом снижаете стабильность сервиса, то ваше улучшение неравномерно и вряд ли полученное преимущество окажется долгосрочным и устойчивым. Важно следить за всеми четырьмя ключевыми метриками. Трансформации с предсказуемой и долгосрочной ценностью — это такие, которые положительно влияют *на все метрики*.

Итак, мы узнали, откуда берутся наши метрики. Усложним картину, сопоставив общую схему с реальным процессом доставки ПО в организации. В следующем разделе я покажу, как выполнить этот «рефакторинг схемы».

Рефакторинг схемы

Важно дать определение каждой метрики *для конкретных условий*. Как вы, наверное, догадались, первые две метрики измеряют то, что происходит в CI-пайплайнах, а вторая пара отслеживает простои сервиса и его восстановление.

Будьте предельно внимательны, когда будете определять область применения при рефакторинге схемы. Какие изменения вы анализируете — для всего ПО в организации или только те, что относятся к вашей части ПО? Учитываете ли вы изменения всей инфраструктуры или только те, что происходят в ПО и сервисах? Все эти варианты возможны, но следует помнить, что *область применения должна быть одинакова для всех четырех метрик*. Если при расчете времени выполнения изменений и частоты развертываний вы учитываете изменения инфраструктуры, то учитывайте и сбои, вызванные этими изменениями.

---

[1] Исправление не обязательно означает код. Поскольку речь идет о возобновлении работы сервиса в принципе, то для остановки отсчета времени восстановления вполне подходит и, к примеру, автоматическое переключение на резервный узел.

Пайплайны как первая отправная точка

Какие пайплайны следует брать в расчет? Те, которые:

- следят за изменениями кода и конфигурации в репозитории исходного кода в рамках выбранной области применения;
- в ответ на эти изменения выполняют различные действия (например, компиляцию, автоматическое тестирование и сборку);
- развертывают результаты на продакшен.

Не стоит учитывать задачи непрерывной интеграции, которые выполняются для других целей, например резервное копирование баз данных.

Если в вашей организации один репозиторий кода обслуживается единым сквозным пайплайном (например, монолитное приложение в монорепозитории, которое собирается и напрямую развертывается на продакшен одним набором действий), то задача значительно упрощается. Эта модель показана на рис. 1.2.

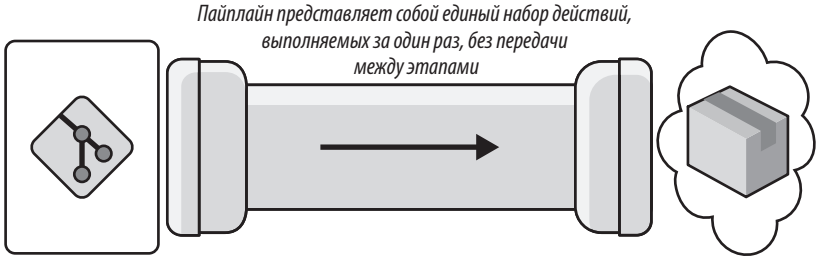

Рис. 1.2. Простейшая модель «система управления версиями — пайплайн — развертывание»

К сожалению, хотя эта модель в точности соответствует нашей базовой схеме, на практике я редко с ней сталкивался. Скорее всего, чтобы схема отражала ваши реалии, потребуется более глубокий ее рефакторинг.

Следующий по простоте измерения и первый значительный шаг в рефакторинге схемы — это набор сквозных пайплайнов, по одному на каждый артефакт или репозиторий (например, на каждый микросервис), где каждый пайплайн выполняет всю необходимую работу и завершается развертыванием на продакшен (рис. 1.3). Например, если вы используете Azure DevOps, создать такие пайплайны несложно[1].

[1] Более того, именно эту модель рекомендует использовать Microsoft.

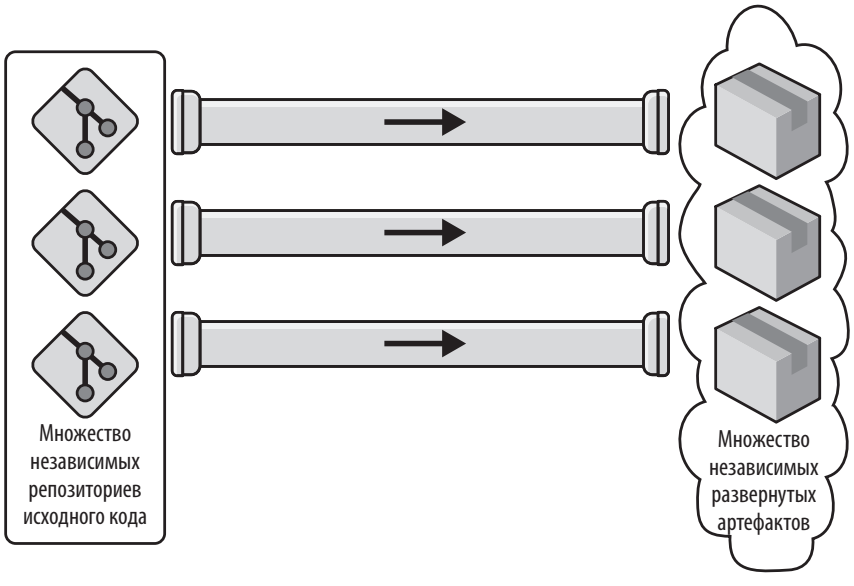

Рис. 1.3. Модель «множества сквозных пайплайнов» идеально подходит для микросервисов

Эти первые две конфигурации пайплайнов, вероятно, *похожи* на ваши, но я полагаю, что ваша картина будет несколько сложнее и потребует еще одного рефакторинга для разделения на серию суб-пайплайнов (рис. 1.4). Рассмотрим пример с тремя такими суб-пайплайнами, которые соединяются в сквозной процесс для доставки изменения на продакшен.

Первый суб-пайплайн может отслеживать пуши в репозитории, выполнять компиляцию, упаковку, модульное и компонентное тестирование, а затем публиковать результат в репозиторий бинарных артефактов. За ним может следовать второй, независимый суб-пайплайн, который развертывает этот новый артефакт в одну или несколько тестовых сред. Третий суб-пайплайн, который может запускаться процессом, подобным CAB[1], в итоге развертывает изменение на продакшен.

[1] Аббревиатура CAB расшифровывается как «change-advisory board» (совет по утверждению изменений). Самый известный пример такой группы, которая регулярно собирается для утверждения релизов кода и конфигурации, описан в ставшей классической книге Джина Кима, Кевина Бера (Kevin Behr) и Джорджа Спаффорда (George Spafford) «The Phoenix Project» (IT Revolution Press, 2018, https://oreil.ly/b5609) («Проект “Феникс”. Как DevOps устраняет хаос и ускоряет развитие компании»).

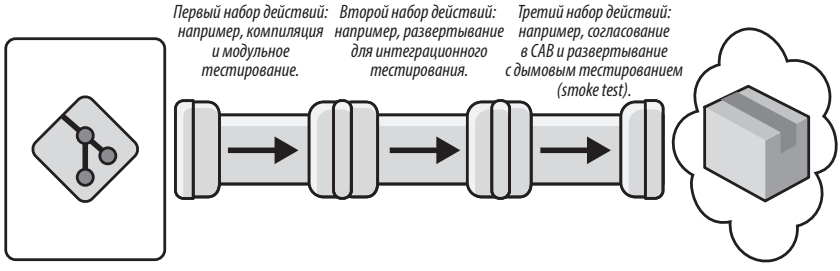

Рис. 1.4. Модель «пайплайн из множества суб-пайплайнов» — это та, которую я встречаю чаще всего

Надеюсь, вы выбрали подходящий для вас вариант. Если же нет, то есть еще и четвертая основная разновидность пайплайнов, к которой мы придем на последнем шаге рефакторинга схемы: многоступенчатый пайплайн слияния (fan-in), показанный на рис. 1.5. Для первой стадии в этой модели обычно используются отдельные суб-пайплайны (по одному на репозиторий), которые затем «сходятся» к общему суб-пайплайну или набору суб-пайплайнов, доводящих изменение до продакшена.

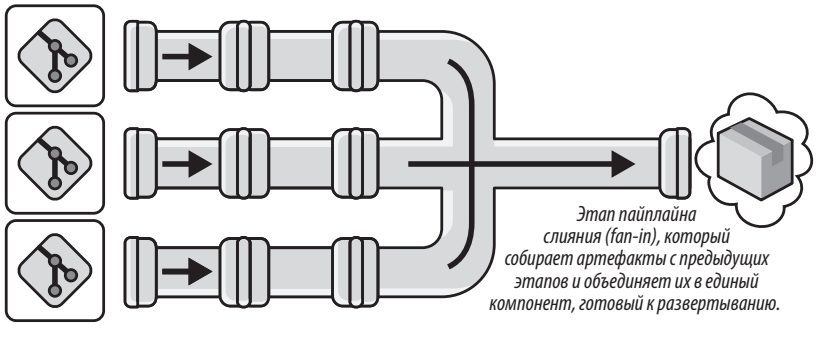

Рис. 1.5. Модель многоступенчатого пайплайна слияния (fan-in)

Определение точек сбора данных

Помимо четырех метрик, у нас есть четыре точки сбора данных. Определим их расположение в нашей схеме, какой бы она ни была. До сих пор мы говорили о пайплайнах, поскольку они обычно предоставляют две из этих точек: метку времени коммита и метку времени развертывания. Третья и четвертая точки сбора данных берутся из меток времени, создаваемых в момент обнаружения деградации сервиса и в момент, когда эта деградация помечается как «устраненная». Теперь обсудим подробнее каждую из них.

Метка времени коммита

При рассмотрении практики работы команд неизбежно возникают тонкие нюансы. Используют ли команды ветвление по фичам? Применяют ли пул-реквесты? Может быть, они используют разнообразные подходы? В идеале (как предлагают авторы «Accelerate») отсчет времени должен начинаться в момент, когда любой набор изменений разработчика считается завершенным и сделан его коммит — независимо от того, где это происходит. Если команды работают именно так, будьте внимательны: удержание изменений в ветках не только удлиняет циклы обратной связи, но и создает дополнительные накладные расходы, а также предъявляет повышенные требования к инфраструктуре. (Об этом я расскажу в следующем подразделе.)

Из-за этой специфики некоторые команды предпочитают использовать запуск пайплайна после слияния в основную ветку (main) в качестве опосредованной точки отсчета или метки времени коммита. Я понимаю, что такая точка отсчета может выглядеть как «костыль»[1], но если вы ее выберете в качестве триггера, будьте, кроме того, готовы к угрызениям совести — за то, что отклоняетесь от лучших принятых практик. Независимо от того, учитываем мы это дополнительное время ожидания или нет, метрики принесут множество других преимуществ, даже если вы позволите себе упростить процесс и начнете отсчет, когда код попадет в основную ветку (main). Если (и когда) эти методы действительно значительно ухудшат процесс доставки, в книге «Accelerate» вы найдете другие рекомендуемые подходы (например, парное программирование и магистральная разработка[2] (TBD, trunk-based development)[3], которые влияют на определение метки времени коммита, предлагая брать за старт таймера время попадания изменения в основную ветку. К тому моменту вы уже начнете видеть пользу от метрик и захотите усовершенствовать процесс их сбора.

Метка времени развертывания

Разобравшись с меткой времени коммита, вы вздохнете с облегчением, узнав, что момент «остановки» часов определить гораздо проще: это за-

---

[1] Особенно если вы используете долгоживущие ветки или пул-реквесты, которые подолгу не закрываются, но я уверен, что вы и так о них знаете, и измерить их влияние по отдельности тоже несложно.

[2] Говорят также «ветвление по модели TBD». — *Примеч. науч. ред.*

[3] См. документацию (https://oreil.ly/L5cs0), чтобы узнать все необходимое о TBD. Исходное определение парного программирования можно найти на ресурсе, посвященном экстремальному программированию (https://oreil.ly/pGAfY).

вершение работы пайплайна, выполняющего финальное развертывание на продакшен. Но нет ли тут некоторой поблажки тем, кто проводит ручное дымовое тестирование уже *после* развертывания? Да, возникает, но, повторюсь, это остается на вашей совести. Если же вам действительно необходимо это финальное действие, вы всегда можете добавить контрольную точку в конце пайплайнов, которую вручную зафиксирует QA-инженер или другой специалист, отвечающий за проверку развертываний, как только убедится в успешности развертывания.

Сложности, характерные для многоэтапных пайплайнов и пайплайнов слияния

Имея эти два источника данных, можно вычислить нужную нам информацию о работе пайплайнов, а именно *общее время выполнения*: интервал между запуском и остановкой таймера. Если у вас реализован один из простейших сценариев работы пайплайна, рассмотренных ранее, — тех, где не используется слияние потоков изменений, — то это относительно легко. Проще всего тем, у кого развернут один или несколько сквозных пайплайнов[1].

Если вам не повезло и у вас есть несколько суб-пайплайнов (как показано на рис. 1.4), вам потребуется собрать дополнительные данные для набора(-ов) изменений, которые были включены в метку времени «старта» и «развертывание» в продакшен. Имея эти данные, вы можете выполнить обработку для расчета общего времени выполнения каждого отдельного изменения.

Если вы используете пайплайн слияния (как на рис. 1.5), эта обработка, вероятно, будет сложнее. Почему? Как видно на рис. 1.6, вам нужен способ определять, откуда берется номер развертывания для изменения 264 — из Репозитория A, Репозитория B или Репозитория C), — чтобы получить метку времени «старта» для этого изменения. Если ваше развертывание объединяет несколько изменений, вам придется отследить каждое из них отдельно, чтобы получить его время «старта».

Очевидно, что в любом случае, независимо от сложности ваших пайплайнов, учитывать следует только те сборки, которые развертывают обнов-

---

[1] Если это соображение навело вас на мысль, что наличие нескольких независимых пайплайнов — по одному на артефакт — это хорошо, примите мои поздравления: вы сами себе напомнили об одном из ключевых принципов микросервисов — независимости развертывания. Если же оно заставляет вас тосковать по монолиту, вспомните о других преимуществах микросервисов, часть из которых мы рассмотрим в конце этой главы.

ления сервисов для пользователей. Убедитесь, что вы измеряете именно такие[1].

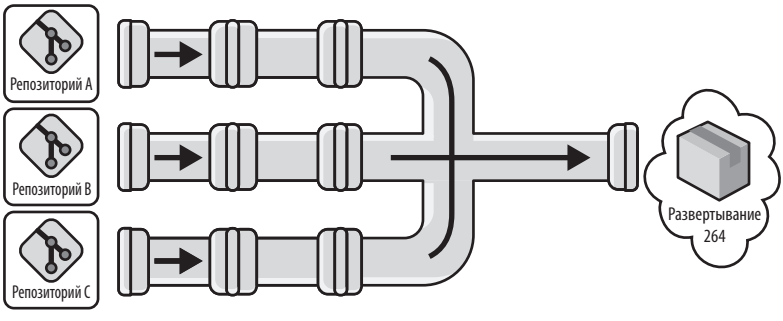

Рис. 1.6. Определение точек сбора данных в модели «пайплайна слияния»

Перед тем как идти дальше, стоит затронуть еще один, заключительный вопрос сбора данных из пайплайнов: какие именно запуски пайплайнов учитывать? «Accelerate» снова не дает четких указаний на этот счет, но для вас должны быть важны только успешные запуски. Если сборка начинается, но падает на этапе компиляции, это искусственно улучшит ваши показатели времени выполнения, поскольку в статистику попадет очень «быстрая» сборка. Если вам захочется манипулировать метриками (а ключевое достоинство четырех основных метрик состоит как раз в том, что ими, насколько мне известно, нельзя манипулировать), то достаточно просто отправлять множество заведомо сломанных сборок, желательно таких, которые завершатся *очень* быстро.

Мониторинг сбоев сервисов

Хотя измерения, связанные с пайплайнами, выполнять относительно просто, третий и последний источник исходных данных можно интерпретировать гораздо шире.

Основная сложность заключается в определении самого понятия «сбой на продакшен». Если сбой произошел, но его никто не обнаружил, можно ли считать, что он вообще был? При работе с четырьмя ключевыми

[1] Иногда спрашивают: «А как быть со сборками инфраструктуры?» Их изредка включают в расчет «четырех ключевых метрик», но я бы не стал настаивать на их обязательном учете. А как насчет пайплайнов, запускаемых по расписанию, а не по изменению? Не учитывайте их. Поскольку по итогу их работы ничего не меняется, они не приводят к развертыванию.

метриками я всегда отвечаю на этот вопрос отрицательно. Я определяю «сбои на продакшен» как любые ситуации, когда потребитель сервиса становится неспособным или даже теряет желание оставаться в системе для завершения начатой задачи. Косметические дефекты сбоями сервиса не считаются, но если «работающая система» работает настолько медленно, что вызывает аномальный отток пользователей, то она явно переживает сбой. Здесь присутствует элемент субъективной оценки, и это нормально: выберите определение, которое вас устраивает, и последовательно его придерживайтесь, оставаясь честным перед собой.

Теперь необходимо фиксировать сбои сервисов — это третья и последняя точка сбора данных, которую обычно реализуют с помощью тикета «сбой изменения». Создание такого тикета дает метку времени начала нового отсчета, а его закрытие — соответствующую метку окончания. Время начала, время окончания и количество тикетов — это все точки исходных данных, которые потребуются для дальнейших расчетов. Тикет следует закрывать, когда работа сервиса восстановлена. Восстановление может не означать устранение первопричины сбоя, и это нормально, поскольку важно обеспечить стабильность сервиса. Откат изменений, чтобы вернуться в онлайн и обслуживать клиентов, является приемлемым решением в данном контексте[1].

Но как быть, если у вас нет продакшен-среды? Во-первых, разве вы еще не пытались начать переход к непрерывному развертыванию? Уже явно пора. Во-вторых, продакшен-среда доступна не всем. Хотя это и не очень хорошо, но даже в таких условиях можно использовать четыре ключевые метрики. Для этого необходимо определить наивысший доступный уровень вашей среды, то есть той общей среды, в которую доставляются все изменения команды и которая ближе всего к продакшену. Вероятно, она называется средой системной интеграции, предпродакшен или промежуточной средой (стейджинг). Главное, что, принимая изменения для этой среды, вы исходите из того, что для их переноса на продакшен дополнительно ничего делать не нужно.

---

[1] Кто-то опять укажет, что время открытия тикета не совпадает с моментом возникновения сбоя сервиса. Это верно. Возможно, вам стоит привязать систему мониторинга к созданию таких тикетов, чтобы обойти эту проблему. Если у вас есть такая возможность, поздравляю: вы, вероятно, находитесь на этапе «тонкой настройки» внедрения четырех ключевых метрик. Большинству команд, особенно в начале, такая точность только снится, поэтому для старта будет достаточно и тикетов, заведенных вручную.