

12+
СЕРГЕЙ КИРНИЦКИЙ

Микросервисы без иллюзий

АРИФМЕТИКА АРХИТЕКТУРНОГО
РЕШЕНИЯ



Сергей Кирницкий

**Микросервисы без
иллюзий. Арифметика
архитектурного решения**

«Издательские решения»

Кирницкий С.

Микросервисы без иллюзий. Арифметика архитектурного решения
/ С. Кирницкий — «Издательские решения»,

Микросервисы давно считаются признаком серьёзной системы. Эта книга показывает, что они родились как организационный инструмент: способ дать сотням инженеров работать, не блокируя друг друга. Автор прослеживает путь идеи от служебного распоряжения в Amazon до архитектуры небольших стартапов, подсчитывает цену каждой сетевой границы и разбирает опыт компаний, которые эту цену измерили. Для инженеров, архитекторов и руководителей, выбирающих архитектуру.

Содержание

Пролог. Пост о мониторинге	6
Часть I. Как мы сюда попали	9
Глава 1. Проблема, которую решали	10
1.1. Указ Безоса	10
1.2. Netflix после катастрофы	13
1.3. Uber: та же боль в режиме гиперроста	15
1.4. Закон Конвея и арифметика Брукса	18
1.5. Общая черта	21
Глава 2. Как решение мутировало в рецепт	24
2.1. Формализация	24
2.2. Survivorship bias	27
2.3. Книги без контекста	29
2.4. Тетралогия хайпа	31
2.5. Как исчезают оговорки	34
Глава 3. Почему это прижилось	37
3.1. Карьерные стимулы	37
Конец ознакомительного фрагмента.	41

Микросервисы без иллюзий Арифметика архитектурного решения

Сергей Кирницкий

Иллюстрация на обложке С использованием ChatGPT

© Сергей Кирницкий, 2026

ISBN 978-5-0071-4407-0

Создано в интеллектуальной издательской системе Ridero

Пролог. Пост о мониторинге

22 марта 2023 года в инженерном блоге Prime Video Tech вышел текст «Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%». Речь в нём шла о том, как в видеосервисе Amazon масштабировали мониторинг звука и видео и сократили его стоимость на 90%. Автор, инженер Prime Video Марцин Кольны, рассказывал о работе одной команды. Это была команда анализа качества видео (Video Quality Analysis): она следила за потоками, которые в этот момент смотрят зрители. Инструмент искал в них заморозки кадра, повреждённые блоки изображения, рассинхронизацию звука и видео. Пока он работает исправно, зритель о нём не знает.

Команда сообщила, что заменила одним процессом распределённые бессерверные (serverless) компоненты, из которых был собран инструмент, и что расходы на инфраструктуру снизились более чем на 90%. Остальное в посте — подробности: задача, решение, результат. Это обычный жанр инженерных блогов, и читают такие тексты коллеги с похожими задачами. Один инструмент, одна команда, одно число.

Широкое обсуждение началось через шесть недель: 3 мая 2023 года о посте написало издание InfoQ. На следующий день, 4 мая, вышли материалы The New Stack и The Stack. The New Stack назвал свою заметку «Return of the Monolith: Amazon Dumps Microservices for Video Monitoring» — «Возвращение монолита: Amazon отказывается от микросервисов для мониторинга видео». Заголовок The Stack сообщал, что сервис Prime Video отказывается от микросервисов и сокращает счёт за облачную инфраструктуру на 90%. Число 90% взято из заголовка поста без изменений.

В тот же день ссылка на сам пост попала на Hacker News и собрала 989 баллов и 507 комментариев. Тогда же свой текст опубликовал Дэвид Хайнмайер Ханссон, создатель Ruby on Rails: «Even Amazon can't make sense of serverless or microservices» — «Даже Amazon не может разобраться в бессерверных вычислениях и микросервисах». Публикация, полтора месяца стоявшая в инженерном блоге, за два дня перешла в новостные издания, на форумы и в личные блоги.

5 мая 2023 года ответил технический директор Amazon Вернер Фогельс. В своём блоге All Things Distributed он опубликовал текст «Monoliths are not dinosaurs» — «Монолиты — не динозавры». Он возражал представлению о монолите как о пережитке прошлого. Фогельс напомнил, что программу, в отличие от моста, можно перестраивать и после того, как она построена. Архитектуру, писал он, стоит пересматривать непредвзято и всякий раз, когда система вырастает на порядок, а единого стиля для всех задач не существует. В пример он привёл тот же Prime Video: трансляции матчей американского футбола Thursday Night Football работают на распределённой архитектуре.

Свою позицию Фогельс сформулировал так: «Строить программные системы, способные развиваться, — это стратегия, а не религия» (Building evolvable software systems is a strategy, not a religion). За три дня, с 3 по 5 мая, обсуждение прошло путь от заметки в профессиональном издании до блога технического директора всей компании. Поводом был сервис мониторинга одной команды.

Факт, который стоит зафиксировать сразу: пост описывал один инструмент, а не архитектуру Prime Video. Слово «монолит» в тексте действительно встречалось и относилось к инструменту: один из разделов назывался «From distributed microservices to a monolith application» — «От распределённых микросервисов к монолитному приложению». Но в том же посте команда писала, что микросервисы и бессерверные компоненты работают на большом масштабе. Выбор между ними и монолитом, по её словам, нужно делать для каждого случая отдельно.

На одном конце этой истории — пост об инструменте, который ищет заморозки кадра. На другом — заголовок о том, что Amazon отказывается от микросервисов. Оговорка «for Video Monitoring» в заголовке The New Stack сохранилась, но стоит в самом конце, после слов об отказе. Оба текста описывают одно и то же событие.

Уточнение по первоисточнику реакцию не отменяет: она факт сама по себе. У неё есть свои даты, свои числа и свои участники, и задокументирована она не хуже самого поста. Технический текст о внутреннем инструменте не становится мировой новостью, если не попадает в давно назревшее напряжение. Пост его не создал, а обнаружил. В мае 2023 года мы обсуждали микросервисы как таковые.

Объяснения такому отклику в посте нет. Он не сообщал о сети ничего нового и не претендовал на это. Сетевой вызов (network call) был дороже вызова функции (function call) задолго до 2023 года; задержки, потери пакетов и частичные отказы появились не тогда. Отсюда вопрос: почему одно число из одного блога прозвучало как откровение, если физика сети не менялась с 1990-х?

В таких спорах под словом «микросервисы» часто смешиваются две разные вещи. Первая из них — контейнеризация, и она не предмет критики этой книги. Docker решает реальную и давнюю проблему: программа работает на машине разработчика и не работает на сервере, потому что окружение там другое. Контейнер упаковывает приложение вместе с его окружением, и главная причина расхождения между машинами уходит. Монолит в контейнере получает изоляцию, воспроизводимость и простое развёртывание. Контейнер меняет способ поставки программы, а не её устройство: ни одного сетевого вызова внутри неё от упаковки не появляется.

То же относится к соседним технологиям. Kubernetes — оркестратор: он перезапускает упавшие процессы, добавляет копии под нагрузкой и разворачивает новые версии, и единое приложение получает всё это наравне с сотней сервисов. Монолит в Kubernetes остаётся монолитом. Облако — место, где работает система, а не способ её разрезать.

Есть и задачи, распределённые по природе. Сеть доставки содержимого (CDN) держит копии данных ближе к пользователям на разных континентах; мессенджер доставляет сообщения между устройствами в разных концах мира; геораспределённая база хранит данные в нескольких регионах сразу. Распределённость таких систем задана самой задачей, и книга с ней не спорит.

Вторая вещь — архитектурное дробление: разбиение единой бизнес-логики на десятки сервисов, которые общаются по сети. Она и есть предмет книги. Задача, которую можно решить в одном процессе, добровольно превращается в распределённую. Вызов функции заменяется сетевым вызовом, и вместе с ним меняется природа операции. Внутри процесса вызов либо выполнен, либо нет; по сети у него появляется третий исход: ответ не пришёл, и неизвестно, была ли операция выполнена. Место, где произошла такая замена, книга называет сетевой границей, и без уточнений слово «граница» дальше означает только его.

Две эти вещи легко спутать, потому что в массовую практику они пришли почти одновременно. Docker стал открытым проектом в 2013 году, а статья Джеймса Льюиса и Мартина Фаулера «Microservices» («Микросервисы») вышла в 2014-м. Фраза «мы переходим на Docker» незаметно превращалась в «нам нужно 30 сервисов», хотя одно из другого не следует. Каждый шаг на этом пути выглядел разумным для тех, кто его делал. Механизм, которым упаковка обростала сетевыми границами, — такой же предмет книги, как само дробление.

Различить контейнеризацию и дробление в собственной системе можно без специальных средств. Монолит в Docker — контейнеризация; 30 сервисов с Kafka между ними — распределённая система. В первом случае одна пользовательская операция остаётся цепочкой вызовов функций внутри процесса и может завершиться одной транзакцией базы. Во втором та же операция проходит через несколько сервисов, у каждого своя база, и сообщение в шине ста-

новится частью бизнес-логики. Kafka здесь упомянута не в упрёк: шина хорошо делает свою работу. Но её появление внутри единой бизнес-логики означает, что сетевых границ в системе уже много.

У этого различия есть оговорка. Единое приложение, которое ходит в базу данных по сети, уже распределено. Но распределённость в нём сосредоточена на одной границе, между приложением и базой, и эту границу закрывает зрелый транзакционный движок. Дробление размножает такие границы. Поэтому сравнивать системы имеет смысл не по числу контейнеров, а по числу мест, где вызов функции стал сетевым.

У разговора о дроблении три стороны: происхождение нормы, её стоимость и опыт тех, кто её проверил. Происхождение — это история о том, как решение организационной проблемы нескольких компаний стало нормой для всех. У стоимости есть счёт за каждую сетевую границу, строка за строкой: время, согласованность, отказы, инфраструктура, люди. Опыт — свидетельства команд, которые посчитали эту стоимость на собственных системах.

Ни одна из этих сторон не начинается в 2023 году. Заголовки мая сообщали о посте, но не объясняли, почему его прочитали именно так: они сами были частью отклика. Чтобы понять реакцию на пост 2023 года, нужно вернуться на двадцать лет назад, к одному служебному распоряжению в Сиэтле.

Часть I. Как мы сюда попали

История когнитивной ошибки: как решение организационной проблемы нескольких компаний стало техническим стандартом целой индустрии.

Между служебным распоряжением в Сиэтле начала 2000-х и архитектурой стартапа из восьми человек лежит несколько пересказов. Распоряжение решало задачу людей, а не кода. Фредерик Брукс ещё в 1975 году показал, что усилия на попарные согласования в группе из n человек растут как $n(n-1)/2$. Когда практику описали, в описании было и предупреждение о цене. Рынок взял перечень характеристик, а предупреждение осталось в тексте.

Никто на этом пути не лгал. Каждый честно пересказывал предыдущий шаг, только короче, и на каждом шаге терялись условия, а выводы сохранялись. Команда из восьми человек, выбравшая десятки сервисов, поступала разумно внутри системы стимулов, которая вознаграждала сложность; сумма разумных решений дала результат, которого никто не выбирал. К 2018—2020 годам открытых сомнений стало больше, но спорили на языке вкусов, а не чисел. Идея была верной для своих авторов. Ошибка возникла в переносе.

Глава 1. Проблема, которую решали

В 2006 году Вернер Фогельс, технический директор Amazon, рассказал журналу ACM Queue о главной архитектурной перемене компании за пять лет. Amazon ушёл от двухуровневого монолита к распределённой платформе сервисов и называл это сервис-ориентированной архитектурой. Слова «микросервисы» в том разговоре нет. Практика оказалась старше термина, которым её потом назовут.

Сегодня мы говорим о микросервисах языком технологий: протоколы, контейнеры, оркестрация, развёртывание. Сама практика родилась из другой задачи — не из производительности кода, а из координации людей. То, что позже стало техническим стандартом, началось как организационный инструмент: способ дать сотням инженеров работать, не блокируя друг друга. У Amazon первым документом этой истории был не архитектурный проект, а распоряжение о том, как командам работать друг с другом.

Три ранние истории — Amazon, Netflix и Uber — сходятся в этом, хотя мотивы у них разные. Amazon принял управленческое решение сверху. У Netflix организационные причины шли рядом с техническими — надёжностью и масштабом трафика. Uber дробился стихийно, без общего плана, со скоростью собственного найма. Форма мотива разная, предел один: число людей, которым приходится согласовывать изменения друг с другом. Цену дробления ни в одном из трёх случаев не скрывали — её описывали те, кто её платил.

У этого предела простая арифметика: согласований становится больше гораздо быстрее, чем людей. Каждое изменение общей схемы данных, каждое общее развёртывание, каждый спор о том, чья правка сломала сборку, — это разговор со всеми, кого изменение задевает. На каком-то масштабе граница сервиса перестаёт быть только техническим решением и становится границей переговоров. Команды договариваются о контракте один раз, а не о каждом изменении. Такая граница покупает время, которое иначе ушло бы на согласования. Платить за неё приходится всем, кто её провёл, а покупка есть не у каждого.

1.1. Указ Безоса

Имя главного приложения Amazon видели даже покупатели: оно стояло в адресах страниц магазина. Приложение называлось Obidos. Фогельс в том же интервью 2006 года рассказывал, что компания начиналась с одной программы на веб-сервере, которая обращалась к базе данных. Со временем Obidos вообрал всю бизнес-логику, всё отображение и все функции, которыми магазин стал известен, — похожие товары, рекомендации, отзывы. Для молодого магазина такое устройство разумно: одна программа, одно развёртывание, всё в одном месте.

К 2001 году, по словам Фогельса, стало ясно, что приложение, собиравшее страницы сайта, больше не масштабируется. Дело было не только в нагрузке. Баз данных к тому времени было несколько, и все они оставались общим ресурсом, который мешал наращивать бизнес. Фогельс объяснял это прямо: и сайт, и внутренние процессы не могли развиваться свободно, потому что ими пользовалось множество разных команд. Техническое ограничение и организационное здесь совпали: общее было трудно и масштабировать, и менять.

Как это выглядело изнутри, публичные источники описывают скупо. Роб Бригхэм из AWS на конференции re: Invent 2015 года называл сайт Amazon 2001 года большим архитектурным монолитом. В его описании части системы были так тесно связаны, что вели себя как одно целое, и каждое изменение уходило в боевую среду только вместе со всем приложением.

Точного числа инженеров тех лет компания не публиковала. По пересказу доклада Бригхэма в The New Stack (2015), в 2000 году инженерная группа сводила в одну версию изменения сотен разработчиков. Для них общий код и общие базы означали одно: изменение схемы

задевало чужую работу и требовало согласования всех со всеми. Сотни людей меняли одно приложение и ждали друг друга.

Распоряжение, с которого началась перестройка, никогда не публиковалось. Его знают по пересказу Стива Йегге, который проработал в Amazon около шести с половиной лет, а к 2011 году — столько же в Google. В октябре 2011 года Йегге написал длинный пост в Google+ для коллег, но ошибся с настройками доступа, и текст стал публичным. Автор его удалил, однако копии уже разошлись. Дату распоряжения Йегге называет сам и с оговоркой: около 2002 года, плюс-минус год.

В его пересказе распоряжение Джеффа Безоса требует следующего. Все команды отныне открывают свои данные и функции только через сервисные интерфейсы и общаются друг с другом только через них. Никакое другое взаимодействие между программами не допускается: ни прямого связывания, ни чтения чужих хранилищ данных, ни общей памяти, ни обходных путей. Технология не важна: HTTP, CORBA, публикация с подпиской или собственный протокол. Любой интерфейс с самого начала проектируется так, чтобы его можно было открыть внешним разработчикам, без исключений; кто этого не сделает, будет уволен.

Сам Йегге вводит этот перечень словами «что-то в таком роде» (something along these lines). Это пересказ по памяти спустя девять лет, а не документ. Есть в нём и заключительное пожелание хорошего дня, но его Йегге тут же признаёт своей шуткой, а не словами Безоса. Проверять исполнение, по его словам, Безос поручил группе во главе с Риком Дальзеллом, директором по информационным технологиям.

В те же годы появилась и вторая мера — команда на две пиццы. Колин Брайар и Билл Карр в книге 2021 года рассказывают о её происхождении так: Безос предложил не управлять зависимостями между командами, а убрать их. Реализацию получил тот же Дальзелл, и он вернулся с моделью, в которой команда не больше, чем можно накормить двумя большими пиццами. Журналист Брэд Стоун в 2013 году называл такие команды автономными группами меньше десяти человек и относил объявление реформы к началу 2002 года.

В распоряжении нет ни слова о том, как писать код внутри команды. Язык, хранилище и протокол команда выбирает сама; пересказ прямо говорит, что технология не важна. Регулируется другое: кто с кем может разговаривать и каким способом. Единственный разрешённый канал между командами — сетевой вызов (network call) через опубликованный интерфейс. Это управленческий документ, написанный языком архитектуры.

Брайар и Карр передают замысел Безоса почти теми же словами: команды должны взаимодействовать через машины и чётко определённые интерфейсы, а не через людей, письма и совещания. Фогельс в 2006 году описывал ту же перемену с другой стороны. Решение он называл итогом серьёзного самоанализа: компания пришла к выводу, что сервисы дадут ей нужную степень изоляции. По его словам, у каждого сервиса есть своя команда, и она полностью за него отвечает, включая эксплуатацию: кто сервис построил, тот и держит его в боевой среде. Граница сервиса совпала с границей команды.

Из этого следует главное для всей истории. Сначала решено, что команды будут отделены друг от друга, и только потом — как они будут обмениваться данными. Если читать распоряжение как архитектурный манифест, в нём видна техника. Если читать его как приказ по организации, видно, ради чего эта техника понадобилась.

За следующие пару лет, по свидетельству Йегге, Amazon изнутри превратился в сервис-ориентированную архитектуру. Масштаб перемены виден по числам Фогельса: в 2006 году приложение, собирая главную страницу Amazon.com, обращалось более чем к ста сервисам. В статье 2007 года о хранилище Дунато инженеры Amazon писали о более чем 150 сервисах на один запрос страницы и о сотнях сервисов в целом. На месте одного приложения с общими базами стояла сеть сервисов со своими командами.

Цену Йегге описывает подробно. Заявка о сбое теперь могла пройти через цепочку из двадцати сервисов, прежде чем находился её настоящий владелец. Даже при отклике каждой команды за пятнадцать минут на поиск могли уйти часы, если не построить для этого отдельную систему метрик и отчётов. Любая соседняя команда могла случайно завалить чужой сервис запросами, поэтому квоты и ограничение нагрузки понадобились в каждом сервисе. Мониторинг и контроль качества, по его формулировке, превратились в одно и то же: каждый интерфейс приходится проверять постоянно, прямо в работе. Сотни сервисов нельзя найти без отдельного механизма обнаружения, а отлаживать чужой код без единого стандартного способа запускать любой сервис стало почти невозможно.

Ещё один урок Йегге формулирует не технически. Работа через сервисы научила команды не доверять друг другу почти так же, как не доверяют внешним разработчикам. Это не упрёк командам, а свойство устройства: интерфейс, который в любой момент могут открыть чужим, приходится защищать и от своих. Доверие между соседями сменилось проверкой — квотой, контрактом, наблюдением за каждым вызовом.

Карточка маситаба

Компания, год: Amazon, 2001—2007; распоряжение — около 2002 года, по воспоминанию Йегге (плюс-минус год).

Инженеров: точных данных нет; по рассказу Бригхэма (re: Invent 2015) — сотни разработчиков.

Пользователи / трафик: опубликованных данных нет.

Сервисов (до → после): одно приложение Obidos и несколько общих баз данных → более 100 сервисов на одну страницу (2006), более 150 (2007), сотни в целом.

Что именно изменилось: данные и функции команд доступны только через сетевые интерфейсы; прямой доступ к чужим базам запрещён; команда отвечает за свой сервис и его эксплуатацию; команды меньше десяти человек.

Источник: S. Yegge (2011); J. Gray, W. Vogels, ACM Queue (2006); G. DeCandia et al. (2007); R. Brigham, re: Invent (2015), по пересказу The New Stack (2015); B. Stone (2013); C. Bryar, B. Carr (2021).

Насколько можно доверять пересказу, стоит сказать прямо. Формулировкам — нет: это память одного человека с шуткой автора внутри. Структуре — да, потому что её независимо подтверждает Фогельс: прямого доступа к базе снаружи сервиса нет, данных между сервисами никто не делит. С датой хуже: Фогельс называет 2001 год, когда приложение перестало справляться, а Энди Джесси, руководитель AWS, в 2016 году относил превращение Amazon в сервисную компанию примерно к 2000 году. Надёжно известно одно: в начале 2000-х Amazon сознательно перестроился вокруг сервисов, и решение было принято наверху.

Amazon видел цену и платил её. По словам Йегге, который ушёл из Amazon в середине 2005 года, сервисы там стали строить уже не из страха увольнения, а потому что поняли, что так правильно. Альтернативу компания знала по собственному опыту. Против часов, потерянных на поиске владельца сбоя, стояло ожидание, которое каждое изменение в Obidos проводило в общей очереди. Для той организации и того числа людей сделка окупалась.

При этом сама потребность договариваться никуда не делась. Она переехала из общего кода и общих баз в контракты интерфейсов, квоты, дежурства и мониторинг. Amazon обменял постоянное согласование всего со всеми на редкие и жёсткие договорённости на границах команд.

Требование проектировать любой интерфейс так, будто его однажды откроют внешним разработчикам, со временем перестало быть мысленным упражнением. В ноябре 2004 года Amazon открыл бета-версию очереди сообщений SQS. В марте 2006 года вышло хранилище S3, и в объявлении компания писала, что даёт любому разработчику доступ к той же инфраструк-

туре, на которой работают её собственные сайты. Бета вычислительного сервиса EC2 открылась в августе 2006 года.

Связь распоряжения с AWS держится на свидетельстве одного участника. Йегге в 2011 году писал, что Безос первым понял: инфраструктуру, построенную для продажи и доставки книг и прочих товаров, можно превратить в универсальную вычислительную платформу. Джесси, рассказывая в 2016 году о происхождении AWS, начинал с Merchant.com — проекта около 2000 года, который открывал платформу Amazon другим продавцам. Чтобы запустить проект, систему пришлось разобрать на документированные интерфейсы, и Amazon, по словам Джесси, очень тихо стал сервисной компанией; дальше в рассказе идут трудности с инфраструктурой. О распоряжении Безоса он не говорит, но и его версия выводит AWS из той же дисциплины внутренних интерфейсов.

Термина «микросервисы» у этой истории ещё нет. Amazon не изобретал архитектурный стиль: он решал, кто с кем разговаривает и во что это обходится, и продавать интерфейсы, спроектированные как внешние, начал уже потом. Интерфейсы — побочный продукт разделения людей, а AWS — побочный продукт интерфейсов.

1.2. Netflix после катастрофы

В августе 2008 года у Netflix случилось крупное повреждение базы данных, и три дня компания не могла отправлять подписчикам DVD. Для сервиса, который рассылал диски по почте, это была остановка самой заметной части работы. По сообщениям прессы того времени, сбой пришёлся на 12—14 августа. Причину компания тогда называла лишь технической проблемой, а какая именно система хранения отказала, так и не уточнила. Об инциденте рассказали сами инженеры — Юрий Израилевский, Стеван Влаович и Руслан Мешенберг — в посте «Completing the Netflix Cloud Migration» («Переезд Netflix в облако завершён») в феврале 2016 года. С этого сбоя, по их словам, и начался путь компании в облако.

Вывод из сбоя авторы поста формулируют технически. Компании пришлось уйти от «вертикально масштабируемых единых точек отказа» (vertically scaled single points of failure), таких как реляционные базы в собственном дата-центре. Вертикальное масштабирование означает, что растущую нагрузку берёт на себя всё более мощная машина. Пока такая машина одна, её отказ останавливает всё, что от неё зависит. Замену Netflix искал в горизонтально масштабируемых распределённых системах в облаке, где нагрузку делят много одинаковых машин и потеря одной не видна снаружи. Причина перестройки здесь разумная, техническая и привязанная к конкретному сбою.

Вторая причина — масштаб. По данным того же поста, с 2008 по 2016 год число стриминговых участников выросло в восемь раз, а общий объём просмотра — на три порядка. Годовые отчёты дают близкую картину, хотя считают по-разному: около 9,4 млн подписчиков на конец 2008 года вместе с дисками и 74,76 млн участников одного только стримингового сервиса на конец 2015 года. Само видео шло к зрителям не из облака, а через собственную сеть доставки Netflix — Open Connect. Из этого следует, что рост просмотра в тысячу раз ещё не означает такого же роста нагрузки на серверы. Авторы поста пишут, что с новыми ресурсоёмкими функциями и растущими объёмами данных поддержать такой рост из собственных дата-центров было бы крайне трудно: серверы просто не успевали бы ставить в стойки.

Переезд занял около семи лет. Все клиентские сервисы Netflix перенёс в облако AWS ещё до 2015 года; последними переезжали биллинг и данные клиентов и сотрудников. Порядок, вероятно, не случаен. Сначала переезжало то, что видит клиент, а в конце — деньги и персональные данные, где ошибка при переносе обходится дороже всего. В начале января 2016 года компания отключила последние остатки дата-центров, которыми пользовался стриминговый

сервис; о DVD-бизнесе пост ничего не говорит. Архитектурный итог авторы описали одной фразой: от монолитного приложения к сотням микросервисов.

Пока шёл переезд, Netflix построил инструменты, которые потом стали символами целой эпохи. Первым публично упомянут Chaos Monkey: в декабре 2010 года инженеры Netflix в блоге компании назвали его одной из первых систем, построенных в AWS. Его работа — случайно выбирать и отключать экземпляры сервисов прямо в боевой среде. В июле 2011 года Израилевский и Ариэль Цейтлин подробно описали его вместе с семейством похожих инструментов: система должна переживать отказ машины так, чтобы клиенты этого не заметили. Код Chaos Monkey компания открыла в июле 2012 года.

Между августом 2008-го и июлем 2011-го — три года и противоположное отношение к отказу. В 2008 году повреждение базы данных остановило отправку дисков на три дня. В 2011 году компания сама отключала свои машины в боевой среде и называла такой отказ распространённым, то есть рядовым событием. Это тот же вывод из августовского сбоя, доведённый до конца. Если отказ неизбежен, систему нужно строить так, чтобы она его переживала, и проверять это на настоящей работе, а не на стенде.

В ноябре 2012 года Netflix анонсировал Hystrix — библиотеку, которая, по словам компании, выросла из работы над устойчивостью, начатой командой API в 2011 году. Hystrix реализовал circuit breaker — механизм, который после серии неудачных вызовов перестаёт обращаться к отказавшей зависимости и сразу возвращает запасной ответ, пока она не восстановится. Проще всего представить это на странице из нескольких блоков: если сервис рекомендаций перестал отвечать, страница открывается без персональной подборки, а не зависает целиком. В июне 2013 года появился Zuul — сервис на границе облака, через который проходят входящие запросы. К тому времени Netflix уже системно открывал код своих инструментов: портал открытых проектов на GitHub появился в ноябре 2011 года, а в начале 2013-го программа выступала под именем NetflixOSS. К октябрю 2015 года она насчитывала больше пятидесяти открытых проектов.

Каждый из этих инструментов отвечал на конкретную проблему конкретного масштаба. У Netflix к концу 2015 года были десятки миллионов подписчиков, около 30 млн из них — за пределами США, и требования к доступности, при которых сбой сразу видят миллионы зрителей. У большинства компаний таких условий нет. Chaos Monkey имеет смысл там, где машин так много, что отказ одной из них — вопрос не «если», а «когда»: в системе из трёх серверов случайное отключение одного было бы не проверкой, а потерей трети мощности. Hystrix нужен там, где один запрос проходит через множество зависимостей и отказ одной не должен останавливать остальные. Вместе с открытым кодом индустрия получила не только сами инструменты, но и неявное допущение, что у неё те же проблемы, что у Netflix.

Одна большая точка отказа сменилась множеством мелких: машин, экземпляров и зависимостей стало больше, и каждая из них могла отказать. Чтобы система переживала их незаметно для клиентов, понадобились отдельные инструменты, отдельная работа и привычка нарочно ломать собственную систему. Эту цену компания описывала в тех же постах, где рассказывала об инструментах.

Карточка масштаба

Компания, год: Netflix, 2008—2016; переезд стримингового сервиса в облако — с августа 2008 года до начала января 2016 года.

Инженеров: опубликованных данных нет; сотрудников на полной ставке — 1 644 на конец 2008 года и около 3 500 на конец 2015 года, без разбивки по функциям.

Пользователи / трафик: около 9,4 млн подписчиков на конец 2008 года (диски и стриминг вместе); 74,76 млн стриминговых участников на конец 2015 года; объём просмотра за восемь лет вырос, по данным компании, на три порядка.

Сервисов (до → после): монолитное приложение → сотни микросервисов (формулировка компании, 2016); точного числа компания не публиковала.

Что именно изменилось: стриминговый сервис целиком ушёл из собственных дата-центров в AWS; вертикально масштабируемые базы в дата-центре заменены горизонтально масштабируемыми распределёнными системами; отказ отдельной машины стал штатным событием, которое проверяют в боевой среде.

Источник: Y. Izrailevsky, S. Vlaovic, R. Meshenberg (2016); Netflix, годовые отчёты за 2008 и 2015 годы; Netflix Tech Blog (2010—2013); Netflix, Hystrix README (2018); T. Mauro, NGINX (2015).

Мотив Netflix смешанный, и в этом ценность случая. История начинается не с людей, которые ждут друг друга, а с базы данных, которая отказала. В посте 2016 года причины переезда названы технические: надёжность и масштаб. Организационная сторона видна в другом источнике — в пересказе выступлений Адриана Кокрофта 2014 года, который Тони Мауро опубликовал в блоге NGINX в феврале 2015 года. Там переход описан как смена модели разработки: от одной группы инженеров вокруг монолитного приложения к множеству небольших команд. Это пересказ, а не слова самого Кокрофта, и число инженеров в нём приведено без года.

Надёжность и масштаб трафика весили не меньше, чем независимость команд. Разложить мотив на доли источники не позволяют. Вероятно, два мотива поддерживали друг друга. Системе из множества независимых компонентов нужен владелец у каждого из них, а работу небольших команд проще всего разграничить именно по таким компонентам.

Netflix — честный пример того, что у дробления бывают технические причины. Тезису книги это не противоречит. Книга не утверждает, что таких причин не бывает, — она утверждает, что у большинства компаний их нет. Условия, в которых действовал Netflix, к тому же проверяемы. Проверка простая: останавливал ли отказ одного компонента основную работу на дни и растёт ли нагрузка быстрее, чем её удаётся обеспечить закупкой серверов. Там, где такие условия есть, горизонтальное масштабирование покупает именно то, ради чего его выбрал Netflix.

В ноябре 2018 года вышла версия Hystrix 1.5.18, и в описании репозитория появился новый статус. Библиотека больше не развивается и находится в режиме поддержки. Объяснение в том же описании короткое: для существующих приложений Hystrix достаточно стабилен, и Netflix продолжает им пользоваться. Внимание компании сместилось к механизмам, которые подстраиваются под поведение приложения в реальном времени, а не работают по заранее заданным настройкам, — например, к адаптивным ограничениям числа одновременных запросов (adaptive concurrency limits). Для новых внутренних проектов Netflix назвал открытую библиотеку resilience4j.

Флагманский инструмент эпохи устарел у собственного создателя через шесть лет после анонса. Идея при этом не оказалась ошибочной: Netflix от неё не отказался. Устарел конкретный ответ — заранее заданные настройки, выросшие из задач команды API 2011 года. За шесть лет изменились и масштаб, и инфраструктура вокруг, и у того же создателя появился другой ответ на тот же вопрос. Инструменты конкретного масштаба стареют вместе с ним.

1.3. Uber: та же боль в режиме гиперроста

В сентябре 2015 года инженеры Uber начали пост о своей архитектуре с признания, знакомого многим стартапам. Компания начинала с монолита, построенного для одной услуги в одном городе: тогда весь Uber был тарифом UberBLACK, а весь его мир — Сан-Франциско. Ретроспектива компании 2020 года уточняет, что около 2012—2013 годов монолитных приложений было, по сути, два. По другим постам компании 2015—2016 годов, основное было

написано на Python и хранило данные в одной базе PostgreSQL, а диспетчеризация поездок — на Node.js. Для одного города и одной услуги большего и не требовалось.

Мир компании быстро перестал помещаться в один город, и вслед за этим ускорился найм. Мэтт Рэнни, главный системный архитектор Uber, рассказывал в 2016 году, что за полтора года инженерная организация выросла примерно с 200 до 2 000 человек. Эти числа известны по конспектам его доклада; официально компания их не публиковала. Пост 2015 года описывает, что происходило с кодом при таком росте. Компоненты оказались тесно связаны, развернуть код значило развернуть всё сразу, а перед любым изменением требовалось знание, которое передавалось только из уст в уста.

Та же ретроспектива описывает этот момент коротко. Когда инженеров стало не десятки, а сотни и у частей системы появились разные команды-владельцы, монолитная архитектура «связала судьбы команд» (tied the fate of teams together). Судьба здесь вполне конкретная: ошибка одной команды могла задержать выпуск всех остальных, а правка в общем коде требовала знания, которого у новичка не было. Долю новичков показывает простая модель с двумя допущениями: численность росла с постоянным темпом, каждый месяц в одно и то же число раз, а ушедших модель не учитывает. Тогда за полтора года она удвоилась трижды с небольшим, примерно раз в пять-шесть месяцев, и в любой момент около половины инженеров работали в компании меньше полугода. Знание, которое передавалось только в разговорах, при таком темпе не успевало дойти до всех.

Выход, который нашли команды, был самым прямым: новая команда не ждала очереди в общем коде, а заводила свой сервис, который могла развернуть сама. В 2014 году сервисов, по воспоминаниям участников, было от двух десятков до сотни. Пост сентября 2015 года говорит о 500 с лишним и тут же признаёт, что найти среди них нужный стало трудной задачей. В начале марта 2016 года, по словам самого Рэнни в аннотации доклада, в боевой среде работало больше 1 000 сервисов. О двух с лишним тысячах инженеры компании писали уже в начале 2017 года. Сервисы появлялись быстрее, чем их успевали учесть.

Главный источник о цене этого роста — доклад Рэнни «What I Wish I Had Known Before Scaling Uber to 1000 Services» на конференции GOTO Chicago в мае 2016 года. По-русски смысл названия такой: что стоило бы знать заранее, прежде чем Uber дорос до тысячи сервисов. Рэнни пришёл в Uber из Voxer, где был сооснователем и техническим директором. Цену здесь описывает не внешний критик и не историк спустя десять лет, а участник, пока рост ещё идёт. Само название фиксирует порядок событий: знание пришло после сервисов, а не до них.

По конспектам доклада, Рэнни называл и то, что сервисы дали. Команды можно было собирать быстро и запускать независимо; каждая сама отвечала за доступность своего сервиса и выбирала инструмент под свою задачу. В комментарии к записи доклада на форуме Hacker News он писал, что главной движущей силой был не рост трафика, а рост команды и темпа выпуска новых функций. Для организации, которая за полтора года выросла вдесятеро, это была не отвлечённая выгода. Это было условие, при котором новые люди вообще начинали работать.

Цена пришла с той же стороны, что и выгода. Свобода выбирать инструмент добавила к двум исходным языкам Uber ещё два: официальный обзор технологий компании в июле 2016 года называл основными Python, Node.js, Go и Java. Рэнни рассказывал, что из-за этого трудно делиться кодом и трудно переходить из команды в команду, а инженерная культура начинает дробиться вместе с языками. На практике это означает, что библиотеку, уже написанную одной командой, соседней может понадобиться писать заново, а инженер, переходящий в другую команду, отчасти начинает с нуля.

Вторая цена — трассировка (tracing), то есть возможность проследить путь одного запроса через все сервисы, которые он затронул. В конспектах доклада Рэнни эта трудность объясняется так: контекст запроса нужно передавать между сервисами на разных языках.

Инженер Uber Юрий Шкуро в феврале 2017 года подтвердил это уже в блоге компании. Сервисы там были написаны на четырёх основных языках, и эта разнородность, по его словам, сделала внедрение распределённой трассировки намного более трудной задачей. Узнать, через кого прошёл запрос, стало отдельным инженерным проектом.

Самое точное наблюдение доклада касается не технологий. В пересказе конспектов Рэнни описывал микросервисы как способ заменить человеческое общение координацией через API: вместо неловкого разговора с соседней командой можно написать ещё немного кода и не разговаривать вовсе. Граница между сервисами на деле оказывалась границей между людьми. Решения, которые раньше принимались в споре над общим кодом, переходили в другую плоскость: чей сервис главнее, чьи приоритеты важнее, кто кого ждёт. Там же Рэнни говорил об обмене сложности на политику, а политикой считал любое решение, которое нарушает порядок «компании важнее команды, команда важнее собственных интересов».

Вывод Рэнни, в пересказе InfoQ, звучит спокойно: у всего есть цена, и компромиссы стоит выбирать намеренно. Судя по докладу, в Uber многие из них делались по ходу: решать приходилось быстрее, чем обдумывать. В этом разница трёх историй. У Amazon решение было сознательным и управленческим: порядок общения команд задал документ сверху, и сервисы пришли следом. Мотивы Netflix смешанные: отказ базы и рост просмотра шли рядом с независимостью команд. Uber рос стихийно: распоряжения, подобного указу Безоса, источники не описывают, и сервисы множились вслед за командами.

Из трёх мотивов у Uber — почти чисто организационный: трафик здесь на втором плане, а скорость найма обгоняет способность договариваться. Затраты на согласование, как давно подсчитал Брукс, растут быстрее, чем число людей в группе. В большинстве компаний эта арифметика проявляется годами, и её трудно отделить от других причин замедления. В Uber её можно было наблюдать в реальном времени. Каждое удвоение приводило в компанию столько же новых людей, сколько в ней уже было, и всем им нужно было договариваться с теми, кто пришёл раньше. Каждый новый сервис сужал круг тех, с кем команде приходилось согласовывать изменения.

Та же компания позже сделает и обратный шаг. Автор ретроспективы 2020 года, инженер Uber Адам Глак, в июле того года писал примерно о 2 200 критичных сервисах и о том, как компания группирует их в домены. Рост организации в том же посте выглядит как переход от сотен инженеров к тысячам. Дробление и укрупнение оказываются частями одной истории. Uber — редкий случай, когда её можно проследить от начала до конца по текстам самой компании.

Карточка масштаба

Компания, год: Uber, 2014—2017; ретроспектива — 2020.

Инженеров: около 200 → около 2 000 за полтора года (примерно 2014—2016), по словам Рэнни в конспектах доклада; официальных данных нет; к 2020 году — «тысячи», без точного числа.

Пользователи / трафик: всего 1 млрд поездок к концу декабря 2015 года → 2 млрд к июлю 2016 года, по объявлениям компании; данных о нагрузке на серверы нет.

Сервисов (до → после): фактически два монолитных приложения (около 2012—2013) → от 20 до 100, по воспоминаниям участников (2014) → больше 500 (сентябрь 2015) → больше 1 000 в боевой среде (март 2016) → больше 2 000 (начало 2017); около 2 200 критичных (июль 2020).

Что именно изменилось: два монолита — основное приложение на Python с общей базой PostgreSQL и диспетчеризация на Node.js — разделены на сервисы, которые команды создают и развёртывают сами; основных языков стало четыре вместо двух — Python, Node.js, Go и Java; каждая команда отвечает за доступность своего сервиса.

Источник: R. W. Schmidt (2015); E. Haddad (2015); M. Ranney, GOTO Chicago (2016), по конспекту Т. Hoff, HighScalability (2016); тот же доклад на QCon New York (2016), InfoQ; L. Lozinski (2016); E. Klitzke (2016); Y. Shkuro (2017); A. Gluck (2020); J. Clemm (2024).

Граница подтверждённого здесь проходит чётко. Числа сервисов с 2015 года, языки, трудности трассировки и исходные монолиты на Python и Node.js описаны в постах самой компании. Рост с 200 до 2 000 инженеров, мысль о границах между людьми и обмен сложности на политику известны по конспектам HighScalability и InfoQ. Запись доклада существует, но сверенной стенограммы нет, и точные слова Рэнни по конспектам не восстановить. Поэтому в этой истории числа сервисов — факт, а числа людей — свидетельство одного участника.

Рост при этом не был ошибкой, которую потом пришлось исправлять. Компании, которая за несколько лет вышла далеко за пределы одного города, нужно было выпускать новые функции быстрее, чем позволял один общий код. Пост 2015 года прямо называет быстрый рост инженерной команды среди причин перехода. Цена, которую описал Рэнни, была платой за эту скорость.

Взрывной рост инженерной организации не диктует единственного ответа. Google пошёл другим путём — единый репозиторий на всю компанию, — и к этому книга вернётся в главе 9. Uber показывает, как на рост ответила одна компания, а не то, как на него обязан отвечать каждый.

В Uber решение принималось не однажды наверху, а много раз внизу — каждой новой командой. Последствия этих решений компания увидела уже в сумме: в языках, в трассировке, в политике. Сервис позволял команде не договариваться с соседями — и у этой свободы была цена.

1.4. Закон Конвея и арифметика Брукса

В апреле 1968 года журнал Datamation напечатал статью программиста Мелвина Конвея. Заголовок статьи, «How Do Committees Invent?», спрашивает, как изобретают комитеты, а речь в ней идёт о любых группах, которые вместе проектируют систему, — не только о программах. Главный тезис Конвей уместил в одно предложение: организации, которые проектируют системы, «вынуждены создавать проекты, копирующие коммуникационные структуры этих организаций» (are constrained to produce designs which are copies of the communication structures of these organizations). Проще говоря, стыки между частями системы повторяют линии, по которым договариваются друг с другом её авторы. Своего имени у этого наблюдения тогда ещё не было.

Имя ему дал Фредерик Брукс. В 1975 году вышла его книга «The Mythical Man-Month» — «Мифический человеко-месяц», о том, почему людей и месяцы в программном проекте нельзя обменивать друг на друга. Брукс сослался на статью Конвея и назвал её тезис законом Конвея; под этим именем он известен до сих пор. В той же книге есть арифметика, которая объясняет, почему этот закон тем заметнее, чем больше организация. Если каждую часть работы нужно отдельно координировать с каждой другой, пишет Брукс, трудозатраты растут как $n(n-1)/2$, где n — число участников. Трое, по его подсчёту, требуют вдвое больше попарного общения, чем двое, а четверо — вшестеро.

Формулу легко проверить самому. У каждого из n человек есть $n-1$ возможных собеседников, но при таком счёте каждая пара учтена дважды, со стороны каждого из двоих, поэтому произведение делится пополам. Десять человек дают 45 пар, пятьдесят — 1 225, тысяча — 499 500. Когда вместо десяти человек работают пятьдесят, людей становится в пять раз больше, а пар — примерно в 27 раз. Когда вместо десяти работает тысяча, людей становится больше в сто раз, а пар — в 11 100 раз. Число людей растёт линейно, число пар — примерно как половина его квадрата.

Обычно эту величину пересказывают как число каналов коммуникации. Основание для этого дал сам Брукс: в другой главе той же книги он пишет о $(n^2 - n) / 2$ стыках, через которые может идти общение. Но там, где формула появляется впервые, он считает не линии на схеме, а трудозатраты на попарную координацию. Для счёта это безразлично: пар столько же. Для смысла разница важна: каждый канал — не провод, а время людей, которое уходит на то, чтобы договориться. Формула описывает худший случай, когда согласовывать приходится всё со всем, и в этом её польза: она показывает, от чего организация защищается своим устройством.

Ни одна организация из тысячи человек не живёт в худшем случае. Её делят на команды, отделы и направления как раз для того, чтобы большинство из полумиллиона возможных разговоров не понадобилось. Внутри команды люди договариваются постоянно, между командами — реже и через общие планы и правила. Пока код остаётся общим, это деление неполное. Правка в общей таблице базы данных или в общей библиотеке задевает всех, кто от них зависит, и разговор снова идёт со всеми, как бы аккуратно ни была нарисована оргструктура.

Рост числа пар меняет не только количество согласований, но и их характер. В небольшой группе автор изменения знает всех, кого оно заденет, и может сам обойти каждого. Когда людей сотни, он уже не может даже перечислить тех, с кем нужно договориться, и этот список приходится выяснять отдельно. Согласование превращается из разговора в процедуру — с очередью, ожиданием и людьми, которые следят, чтобы никого не забыли.

Где именно координация становится главным узким местом, формула не говорит: она показывает скорость роста, а не порог. Порог приходится оценивать по случаям. Amazon, судя по рассказу Бригхэма, упёрся в него с сотнями разработчиков. Uber, по словам Рэнни, дошёл до него на пути от двухсот к двум тысячам. Из этих случаев следует грубая оценка: где-то между сотнями и тысячами инженеров время на согласования начинает перевешивать время на саму работу. До этого предела медленную разработку можно объяснить многим, после него главное объяснение — очередь из согласований.

На этом масштабе у границы сервиса появляется вторая роль: она начинает работать как граница переговоров. Предмет переговоров у неё один — контракт, то есть перечень запросов, которые сервис принимает, и того, что он обещает в ответ. Всё, что лежит за контрактом, команда меняет сама и никого не спрашивает. Команда, которая считает стоимость доставки, может переписать расчёт, сменить хранилище и перестроить свои таблицы, и пока запросы и ответы остаются прежними, соседям об этом знать не нужно. Договариваться снова приходится, только когда меняется сам контракт, а это случается намного реже, чем меняется код за ним.

Порядок выигрыша видно на модели. Модель: тысяча инженеров в ста командах по десять человек; внутри команды договариваются все со всеми; между командами — только через контракты, и у каждой команды пять соседей, с которыми её связывает общая работа. Внутри команд получается $100 \times 45 = 4\,500$ пар. Между командами — 250 пар соседей, и договариваются в них команды, а не каждый инженер с каждым. Вместо 499 500 возможных каналов остаётся меньше пяти тысяч, и число межкомандных пар растёт вместе с числом команд, а не как квадрат числа людей. Реальные организации устроены сложнее, и модель показывает не их, а порядок разницы.

За этот выигрыш приходится платить. Однажды принятый контракт трудно менять: его изменение снова требует переговоров, теперь уже со всеми, кто им пользуется. Ошибку в проведении границы дорого исправить, потому что по обе её стороны стоят команды со своими планами. Граница переговоров окупается там, где за ней действительно работают разные люди, которым без неё пришлось бы договариваться постоянно. Её цена фиксирована, а выигрыш зависит от того, сколько переговоров она отменяет.

Из этой арифметики видно, почему дробление сработало у Amazon и Netflix. Обе компании не столько нарезали код, сколько провели границы сервисов по границам команд. Если архитектура копирует структуру общения, её можно получить, заранее выбрав эту структуру. Распоряжение Безоса сначала определило, как командам разговаривать друг с другом, и сервисы выросли вдоль этих линий. У Netflix организационная сторона перехода известна только по пересказам, но и там сервисы в итоге разошлись по небольшим командам, а технические причины действовали вместе с этой перестройкой. Брукс вслед за Конвеем сделал из закона практический вывод: если проект системы должен свободно меняться, организация должна быть готова меняться вместе с ним.

Имя этот приём получил позже. В декабре 2010 года Джонни Лерой и Мэтт Саймонс из ThoughtWorks опубликовали в Cutter IT Journal статью «Dealing with Creaky Legacy Platforms» — о том, что делать со скрипучими унаследованными платформами. Обновление старой системы они советовали начинать не с кода, а с перегородок между группами, которые мешают командам работать вместе. Этот ход они назвали обратным манёвром Конвея (inverse Conway maneuver): идти от нужной архитектуры к структуре команд и ждать, что система последует за этой структурой. Широко известным термин стал после того, как ThoughtWorks включила его в июльский выпуск своего Technology Radar 2014 года. Amazon проделал этот манёвр примерно за восемь лет до того, как у него появилось имя.

Второе понятие пришло с другой стороны. В 2019 году Мэттью Скелтон и Мануэл Паис в книге «Team Topologies» («Топологии команд») перенесли на команды понятие когнитивной нагрузки (cognitive load). Его источник — теория психолога Джона Свеллера, где когнитивная нагрузка — общий объём умственных усилий, которые занимают рабочую память человека. Команда, по Скелтону и Паису, тоже способна удерживать в голове лишь ограниченную часть системы. Поэтому подсистема, которой владеет команда, не должна вырастать за пределы этой нагрузки.

Вместе с арифметикой Брукса это понятие ставит границе сервиса два предела. Арифметика Брукса ограничивает её снаружи: чем больше сторон должны договориться об изменении, тем дороже каждое изменение. Когнитивная нагрузка ограничивает её изнутри: команда не может надёжно владеть большим, чем способна понимать. Граница, проведённая по линии между командами, учитывает оба предела сразу. Из этого следует, что главная переменная здесь — не число строк и не трафик, а число команд, которым нужна независимость друг от друга.

Слово «закон» здесь обманчиво. Закон Конвея — эмпирическое наблюдение, а не закон физики. Сам Конвей вспоминал, что сначала отправил статью в Harvard Business Review и журнал её отклонил: автор, по мнению редакции, не доказал свой тезис. Брукс к собственным формулировкам относился так же трезво. Знаменитый закон Брукса — добавление людей в запаздывающий программный проект задерживает его ещё сильнее — он сам предварял словами о возмутительном упрощении. Даже формулировка закона Конвея дошла до читателей с поправкой: Брукс, цитируя статью, написал «системы» там, где у Конвея стоят «проекты».

Проверить закон Конвея исследователи пытались. Одна из самых известных проверок — работа Алана Маккормака, Карлисс Болдуин и Джона Руснака в журнале Research Policy в 2012 году. Авторы проверяли так называемую зеркальность (mirroring) — соответствие структуры продукта структуре организации. Они сравнили пять пар программных продуктов с одинаковым назначением: в каждой паре один продукт делала коммерческая компания с тесно связанной командой, другой — слабо связанное сообщество открытого кода. Во всех парах продукт слабо связанной организации оказался заметно лучше разделён на независимые части. Поэтому, насколько изменение в одном компоненте способно задеть другие, разница, по данным авторов, доходила до шести раз.

Это сильное подтверждение, но подтверждение связи, а не механизма: исследование не показывает, что устройство организации порождает устройство кода. Можно предположить и обратное направление: люди, которые пишут слабо связанный код, легче работают слабо связанными группами. Можно предположить и общую причину — например, сам способ, которым открытые проекты принимают правки от множества независимых участников. Поэтому книга использует закон Конвея как объяснительную модель, а не как доказательство: он объясняет, почему истории Amazon, Netflix и Uber похожи, но не заменяет данных о каждой из них.

У закона есть и обратное прочтение: по архитектуре можно восстановить организацию, которая её создала. Тридцать сервисов с контрактами между ними — рисунок, который по закону Конвея оставила бы организация из многих команд, каждой из которых нужно договариваться с соседями через контракты. Такая организация покупала бы каждой границей то, что граница умеет покупать: право менять код за контрактом без переговоров. Но если архитектура отражает организацию, то 30 сервисов у одной команды из 12 человек отражают организацию, которой не существует.

Цену такая команда при этом платит полностью. Каждую границу по-прежнему нужно описать контрактом, изменения на обоих её концах — согласовать, а сервисы — развёртывать в правильном порядке. Но по обе стороны границы стоят одни и те же люди, и переговоры, от которых граница должна была избавлять, им вести не с кем, кроме самих себя. Граница сервиса без границы команды — это цена без покупки.

1.5. Общая черта

Если поставить рядом числа масштаба из трёх историй, они окажутся разного рода. У Amazon начала 2000-х — одно приложение Obidos, которое меняли все вместе. Работали над ним, по рассказу Бригхэма, сотни разработчиков. У Uber — рост примерно с 200 до 2 000 человек за полтора года, по словам Рэнни в 2016 году. У Netflix число инженеров не опубликовано, зато известно, что с 2008 по 2016 год объём просмотра вырос на три порядка. Два числа из трёх описывают людей, одно — трафик.

Числа о людях можно перевести в меру, которую даёт формула Брукса. Двести человек — это 19 900 возможных каналов, две тысячи — 1 999 000, почти два миллиона. За полтора года людей в Uber стало вдесятеро больше, а каналов — примерно в сто раз. Сотни инженеров Amazon дают десятки тысяч каналов и больше. Сами по себе эти числа говорят только о раз-мере, а не о пороге. Судя по собственным описаниям обеих компаний, к моменту перестройки они упёрлись в предел координации: время всё больше уходило на ожидание согласований, а не на сам код.

Третье число описывает не людей. История Netflix начинается с отказавшей базы, а не с очереди в общем коде. Техническая причина пришла первой, но организационный итог оказался тем же, что у Amazon и Uber.

У такой перестройки была цена, и часть её участники описали сами: поиск владельца сбоя через цепочку сервисов, привычку нарочно ломать собственную систему, трассировку запроса через четыре языка. Другие издержки заложены в самом устройстве. Вызов функции (function call) стал сетевым вызовом с его задержками; изменение, которое укладывалось в одну транзакцию одной базы, стало распределённой транзакцией; данные, жившие в одной таблице, пришлось дублировать между сервисами. Для компаний того масштаба всё это было приемлемой ценой. Они покупали организационную скорость, которую при таком числе людей иначе было не получить.

Покупка у трёх компаний была одна и та же, хотя пришли они к ней разными путями. В Amazon, по словам Фогельса, у каждого сервиса была своя команда, которая отвечала за него целиком, вплоть до эксплуатации. В Uber новая команда заводила свой сервис, чтобы не

ждать очереди в общем коде. У Netflix, вероятно, каждый независимый компонент получал своего владельца — небольшую команду. Во всех трёх случаях, насколько позволяют судить источники, граница сервиса легла по границе команды и стала границей переговоров.

Из этого следует общая черта трёх историй. Микросервисы родились как ответ на человеческую проблему — на число людей, которым нужно менять одну систему, не дожидаясь друг друга. Технические причины, как у Netflix, шли рядом с ней, а не вместо неё. Три случая — не выборка, и закономерность на них не доказать. Они показывают, при каком масштабе решение принимали его авторы, но ничего не говорят о том, окупается ли оно при меньшем.

Типичная инженерная команда продуктовой компании выглядит иначе: в ней десятки инженеров. Прямых отраслевых данных о численности таких команд нет, поэтому это пример, а не статистика. Косвенно его поддерживает Stack Overflow Developer Survey 2025 года: около 38% ответивших работают в компаниях меньше чем на 100 сотрудников, не считая фрилансеров. Это размер компании, а не инженерной команды, и опрос говорит только о тех, кто сам решил в нём участвовать. Но в компании меньше чем на сто человек тысячи инженеров нет по определению. По отчёту JetBrains 2022 года, большинство опрошенных разработчиков работает над проектами в командах из 2—7 человек.

По той же формуле команда от десяти до сорока пяти инженеров — это от 45 до 990 каналов, а не полмиллиона. На таком масштабе автор изменения ещё знает всех, кого оно заденет, и согласование остаётся разговором, не превращаясь в процедуру с очередью. Ближе к сотне инженеров это начинает меняться: каналов становится 4 950, и людей уже делят на несколько команд. Но и тогда каналов примерно в сто раз меньше, чем у тысячи. Предел, до которого дошли Amazon и Uber, лежит выше, там, где инженеров уже сотни, и такая компания к нему только подходит.

Сами Amazon и Uber когда-то тоже были по эту сторону предела. Одно приложение Obidos, обращавшееся к базе данных, и монолит, написанный для одной услуги в одном городе, были разумным выбором, пока людей было немного. Предел появился вместе с ростом, а не с первого дня.

Разница здесь — в масштабе задачи, а не в качестве работы. Небольшая команда тоже спорит о правках, ждёт проверки кода и ломает общую сборку; медленной бывает любая разработка. Но медлительность такой команды складывается из других причин, и очередь из согласований всех со всеми среди них не главная. Распоряжение, которое заставило бы её общаться только через интерфейсы, отменило бы разговоры, которые ей почти ничего не стоят. У такой команды нет проблемы, которую решал Безос.

То, что делало решение точным, — число людей, которым нужна независимость друг от друга, — в самом решении не видно. Снаружи видны сервисы и интерфейсы, Chaos Monkey и Nystrix в открытом коде, AWS как услуга, которую может купить любой. Причина — предел координации — осталась внутри организаций, которые заплатили цену: в их штатном расписании, в ожидании согласований, в очередях на изменение общей схемы.

Это неравенство видно даже по источникам трёх историй. О сервисах и библиотеках компании писали сами — в постах, в описаниях репозиторий, в объявлениях о запуске, с датами и номерами версий. О пределе известно гораздо меньше: ни одна из трёх компаний сама не опубликовала, сколько у неё было инженеров. У Amazon вместо числа — «сотни» из пересказа доклада Бригхэма, Netflix число инженеров не публиковал, а рост Uber известен по конспектам доклада Рэнни. Само распоряжение Безоса стало известно потому, что пост Йегге по ошибке оказался публичным.

Отсюда следует вопрос об условии, на котором решение держалось у Amazon, Netflix и Uber. Сколько у вас инженеров?

В пересказе распоряжения Безоса нет числа инженеров, для которых оно написано; в описании сервиса есть его контракт, но нет размера организации, которая провела эту границу.

Решение нескольких организаций, упёршихся в предел координации, вышло к командам, у которых этого предела нет.

Микросервисы начинались как организационный инструмент — способ дать сотням инженеров работать, не блокируя друг друга. Amazon, Netflix и Uber делали его под собственный предел, каждый по своей мерке, и платили за него сами. Инструмент был создан для операционной. Скальпель в руке хирурга и скальпель в ящике на кухне — один и тот же предмет.

Глава 2. Как решение мутировало в рецепт

Летом 2015 года на сайте Мартина Фаулера с разницей меньше чем в неделю вышли два текста с противоположными заголовками. В первом Фаулер советовал начинать новую систему с монолита. Во втором Стефан Тилков возражал ему на той же площадке. Спорили два инженера с большим опытом, и оба исходили из того, что распределённость стоит дорого. Спор шёл не в закрытой переписке: его мог прочитать любой, кто искал тогда слово «микросервисы».

Предупреждения, выходит, были публичными. И всё же до объявлений о вакансиях и до архитектуры стартапов из нескольких человек дошли выводы, а не условия, при которых эти выводы были получены. Отдельный сервис на каждую возможность бизнеса, своя база данных у каждого сервиса, независимое развёртывание — это дошло. Оговорка о том, кому и за какую цену всё это окупается, осталась в исходных текстах, где её никто не прятал. Авторы писали осторожно; осторожность потерялась по дороге.

Для такой потери не нужны ни злой умысел, ни небрежность. Отрасль видела тех, у кого получилось, и почти не видела тех, у кого нет. У успешных она видела архитектуру десятого года жизни, а не первого. Справочники читали ради того, как это делать, и несколько страниц о том, зачем и для кого, в памяти не задерживались. Слова, которыми обозначали разные границы — модели, команды, процесса, сети, — постепенно стали значить одно и то же.

Проще всего увидеть эту потерю на одном формате — 45-минутном докладе. В нём помещаются схема из сорока сервисов, график задержек, история ночного отказа и того, как с ним справились. Фразе «у нас это окупается, потому что у нас тысяча инженеров и отдельная команда платформы» хватило бы минуты. Но когда время на исходе, эту минуту сокращают первой. Доклад — формат, куда оговорки не помещаются, а в те годы именно доклады звучали громче всего.

2.1. Формализация

С 10 по 25 марта 2014 года на сайте Мартина Фаулера частями выходила статья «Microservices» — «Микросервисы», написанная им вместе с Джеймсом Льюисом. Название было коротким, задача — скромной. Авторы не предлагали новый способ строить системы. Они описывали то, что уже видели у нескольких команд и компаний, и сразу оговаривали: формального определения у этого архитектурного стиля нет, есть общие черты систем, к которым применяют это название. Не у каждой такой системы есть все черты, но авторы ожидали, что у большинства найдётся большинство.

Черт насчитали девять. Приложение собирается из сервисов, а не из библиотек внутри одного процесса: компонентизация через сервисы (componentization via services). Сервисы нарезаются по возможностям бизнеса, а не по техническим слоям, и команда вокруг такого сервиса сквозная — от интерфейса до базы. Команда владеет продуктом всю его жизнь и не уходит, сдав проект: продукты, а не проекты. Логика живёт в самих сервисах, связи между ними остаются простыми — умные конечные точки и простые каналы (smart endpoints and dumb pipes). Дальше шли децентрализованное управление, при котором команды сами выбирают инструменты, и децентрализованное управление данными: каждый сервис распоряжается своим хранилищем. Замыкали перечень автоматизация инфраструктуры, проектирование в расчёте на отказ (design for failure) и эволюционное проектирование, при котором систему меняют по частям.

В самом перечне уже было записано условие, хотя и не под этим именем. Автоматизация инфраструктуры и проектирование в расчёте на отказ — свойства не кода, а организации. Они описывают команды, которые умеют без ручной работы собирать, проверять и выкладывать десятки сервисов и заранее исходят из того, что любой из них может не ответить.

Статья была наблюдением, а не рекомендацией, и авторы понимали разницу. Последний её раздел был озаглавлен вопросом, станут ли микросервисы будущим программной архитектуры, и ответ в нём звучал сдержанно. Опыт, писали Льюис и Фаулер, пока положительный по сравнению с монолитными приложениями, но истинные последствия архитектурных решений часто видны лишь через несколько лет. Для микросервисов, по их словам, «прошло недостаточно времени, чтобы вынести окончательное суждение» (not enough time has passed for us to make a full judgement). Свою позицию они назвали осторожным оптимизмом.

Реже вспоминают, что было дальше. За следующие пятнадцать месяцев Фаулер выпустил на том же сайте четыре коротких текста. Каждый прикладывал к описанию своё условие или называл свою цену.

28 августа 2014 года вышла заметка «Microservice Prerequisites» — «Предпосылки для микросервисов». Её мысль уложена в первые строки: если у команды нет некоторых базовых умений, браться за микросервисы не стоит. Умений три. Нужно быстро получать серверы: новый сервер должен подниматься за часы, и хотя полной автоматизации на старте не требуется, без неё дальше не обойтись. Нужен базовый мониторинг, потому что множество слабо связанных сервисов ломается в боевой среде так, что в тестовой это трудно обнаружить. Нужно быстрое развёртывание: сервисов много, и каждый должен доходить до тестовой и боевой среды без долгих процедур, а конвейер развёртывания, по Фаулеру, должен занимать не больше пары часов.

Заметка не была запретом. Фаулер оговаривал, что те же умения нужны и монолиту, а кроме них нужна культура, в которой разработка и эксплуатация работают вместе. Но порядок был задан ясно: сначала навыки и платформа, потом сервисы. Эти три умения и сегодня можно приложить к своей команде за несколько минут, и ни одно из них не требует знать, как будет нарезано приложение.

13 мая 2015 года вышел самый прямой текст серии — «Microservice Premium», «Премия за микросервисы». Слово «премия» (premium) здесь значит надбавку к цене. Фаулер писал, что микросервисы вносят в систему собственную сложность и она ложится надбавкой на стоимость и риск проекта, причём такой, что часто приводит проекты к серьёзным трудностям. Отсюда главное правило заметки, выделенное в оригинале жирным: даже не рассматривать микросервисы, если система не слишком сложна, чтобы управлять ею как монолитом. Следующая фраза говорила ещё прямее: большинство программных систем следует строить как одно монолитное приложение.

К правилу прилагался график. По горизонтали — базовая сложность системы, по вертикали — продуктивность, две кривые — монолит и микросервисы. Пока система проста, дополнительный груз, которого требуют микросервисы, снижает продуктивность, и их кривая идёт ниже. С ростом сложности продуктивность монолита начинает быстро падать, а у микросервисов, где связанность меньше, падает медленнее, и где-то кривые пересекаются. График схематический: чисел на осях нет, и точку пересечения Фаулер за измерение не выдавал. Внизу стояла подпись, не менее важная, чем сами кривые: умение команды весит больше, чем любой выбор между монолитом и микросервисами.

В той же заметке Фаулер показывал, насколько разные вещи прячутся под одним словом. Среди команд, называвших свои системы микросервисными, он встречал и команду из 60 человек с 20 сервисами, и команду из четырёх человек с 200 сервисами. Он упоминал и то, что Technology Radar компании Thoughtworks к тому времени уже дал имя стремлению строить микросервисы там, где они не нужны, — «Microservice Envy», зависть к микросервисам.

По сути, это одно из первых публичных описаний налога, которым облагается распределённость. У неё есть постоянная цена, и до определённой сложности эта цена не окупается. Фаулер не раскладывал налог по статьям — он показывал его форму.

Через три недели, 3 июня 2015 года, вышла заметка «MonolithFirst» — «Сначала монолит». Фаулер пересказывал в ней опыт, собранный у коллег. Почти все успешные истории микросервисов начинались с монолита, который стал слишком большим и был разделён. Почти во всех известных ему случаях, когда систему с нуля строили как набор микросервисов, она в итоге попадала в серьёзные трудности. Совет из заголовка касался очерёдности, а не запрета: сначала монолит, а разделение — когда он действительно станет слишком большим. Заметка честно называла свой предел: большинство знакомых Фаулера склонялись к этому подходу, но единодушия не было.

Раньше этих трёх заметок, 13 августа 2014 года, вышел текст, который связывал новое слово со старой книгой, — «Microservices and the First Law of Distributed Objects», «Микросервисы и первый закон распределённых объектов». В 2002 году в «Patterns of Enterprise Application Architecture», книге о шаблонах архитектуры корпоративных приложений, Фаулер сформулировал правило, которое назвал первым законом проектирования распределённых объектов: не распределяйте свои объекты. Через двенадцать лет он сам положил этот закон рядом с микросервисами — меньше чем через пять месяцев после того, как статья о девяти характеристиках вышла целиком. Важен здесь сам факт: о цене распределённости один из авторов описания писал задолго до того, как появилось слово «микросервисы».

Предупреждения не были монологом. 9 июня 2015 года, через шесть дней после «MonolithFirst», на том же сайте вышла статья Стефана Тилкова «Don't start with a monolith» — «Не начинайте с монолита». Тилков был твёрдо убеждён, что начинать с монолита, как правило, неверно. Части монолита неизбежно срastaются — через общие доменные объекты, общую модель хранения, общие транзакции, — и разрезать его потом на независимые куски крайне трудно. Поэтому достаточно большую систему, считал он, разумнее делить сразу, при условии что предметная область известна очень хорошо. Это сильный аргумент: цена позднего разреза реальна, и тот, кто откладывает границы, потом платит за них дороже.

Важнее, что в главном Тилков соглашался с Фаулером: не стоит вносить в систему сложность дополнительной распределённости без очень веской причины. Если команда способна построить хорошо структурированный монолит, писал он, микросервисы ей, вероятно, вообще не нужны. И масштаб он оговаривал сам: если систему несколько недель пишут два человека, вполне возможно, что без дробления они обойдутся. Даже сторона, которая спорила за раннее деление, записала условие о размере. Спор шёл о том, с чего начинать, а не о том, есть ли у микросервисов цена: её признавали обе стороны. Несогласие не мешало и совместной работе. В «MonolithFirst» Фаулер благодарил Тилкова за замечания к черновику, а в нынешней версии статьи о девяти характеристиках стоят ссылки на оба текста.

Стоит назвать и пределы самих предупреждений. Это были не исследования, а заметки, основанные на наблюдениях, и Фаулер прямо писал, что свидетельств немного. И сама статья 2014 года не нуждается в защите: девять черт точно описывали практику, которая у своих владельцев окупалась. Но условие в текстах было записано — с датами, под ясными заголовками, в открытом доступе.

Если положить рядом написанное и то, что разошлось по индустрии, разрыв виден сразу. Девять характеристик стали языком, на котором мы обсуждали архитектуру. «Премия» говорила о другом — не о том, как устроена система, а о том, когда её устройство окупается. Предупреждения серии указывали за пределы схемы: на сложность задачи, на зрелость эксплуатации, на умение команды, на то, была ли у системы история до разделения. Ни одно из этих условий нельзя увидеть, глядя на диаграмму сервисов, — только глядя на организацию, которая её рисует.

Разница слышна даже в грамматике. Льюис и Фаулер писали языком наблюдателя: у таких систем, как правило, есть эта черта, и не у каждой есть все. В прочитанном та же черта получила модальное слово «должен»: у каждого сервиса должна быть своя база, каждая команда должна развёртываться независимо, связи между сервисами должны быть простыми. Изменилось одно слово, и описание практики нескольких организаций стало требованием к любой.

Рынок — то есть все мы, кто в те годы проектировал системы, нанимал инженеров и выступал с докладами, — взял девять характеристик и выбросил «премию». Прочитано было не «так делают компании с пятью тысячами инженеров, заплатив за это», а «так правильно». Разрыв возник не в тексте: канон содержал предупреждение.

2.2. Survivorship bias

В 1943 году статистик Абрахам Вальд работал в Statistical Research Group — исследовательской группе при Колумбийском университете, которая решала математические задачи для военных. Среди его работ того года — серия меморандумов о том, как оценивать уязвимость самолёта по повреждениям машин, вернувшихся с заданий. Материал для такой оценки был под рукой: на вернувшемся самолёте можно пересчитать пробоины и отметить, где их больше. Трудность заключалась в том, чего в этом материале не было. Самолёты, получившие попадания в уязвимые места, не возвращались, и их пробоин никто не видел.

Отсюда главный ход рассуждения. Пробоины на вернувшихся самолётах показывают не то, где машину нужно укреплять, а то, куда попадать не смертельно. Если в каком-то месте пробоин почти нет, это не значит, что туда редко попадают: это может значить, что после таких попаданий не возвращаются. Вальд превратил это рассуждение в расчёт, который учитывал машины, не вернувшиеся на базу. Сегодня такое искажение называют ошибкой выжившего (survivorship bias): вывод строится по тем, кто прошёл отбор, а сам отбор незаметно меняет картину.

Историю Вальда знают в основном по упрощённому пересказу. Знаменитая картинка — силуэт самолёта, усеянный красными точками, — восходит к иллюстрации, которую около 2005 года сделал для своих выступлений дизайнер Кэмерон Молл, а версия, разошедшаяся по сети, нарисована в 2016 году. Точки на обеих вымышлены. У самого Вальда были не картинка и не короткий совет, а восемь технических меморандумов с расчётами. Метод разбирали и спустя сорок лет: в 1984 году статистики Марк Мангель и Франсиско Саманьего подробно описали его в журнале Американской статистической ассоциации. Но суть рассуждения пересказ сохранил.

Мы узнавали о микросервисах в основном из докладов и инженерных блогов, и эта выборка устроена так же. Доклад попадает в программу конференции, если в нём есть новизна и масштаб: необычная схема, большие числа, трудная задача, решённая изобретательно. История «мы сделали монолит, и он работает» — не доклад. Программный комитет не делает здесь ничего дурного — он отбирает то, что интересно слушать. Слушатель отдаёт конференции несколько дней и рассчитывает узнать то, чего не знал.

Инженерные блоги подчиняются той же логике: пост пишут, когда есть чем удивить. Из отобранного складывается представление о том, как строят системы, и в это представление попадают только вернувшиеся самолёты. Со сцены рассказывают и о трудностях, но это пробоины на вернувшихся машинах. Отказ, который система пережила, превращается в рассказ о её зрелости и в новый инструмент с открытым кодом. Попадание, после которого проект не вернулся, — остановленная на середине миграция, команда, так и не вышедшая на прежнюю скорость, — рассказа не оставляет.

Успехи публичны ещё и потому, что у компании есть причина о них рассказывать. Инженерный бренд помогает найму: рассказ о сложной системе под большой нагрузкой сообщает кандидатам, что здесь решают интересные задачи. Нередко последний слайд такого доклада — приглашение на работу, а запись выступления работает на имя компании ещё годы.

Неудачи приватны, и тоже по понятным причинам: внутренние разборы, как правило, остаются внутри компании, а их участников связывают соглашения о неразглашении. Представим доклад о том, как полтора года ушло на разделение системы, выпуск новых функций не ускорился, а часть сервисов пришлось склеить обратно. Для компании это репутационный риск, для тех, кто принимал решение, — риск личный, и выйти с таким рассказом на сцену под своим именем согласятся немногие. А когда рассказать уже можно, рассказывать часто некому: авторы решения давно уволились, и история ушла вместе с ними. Подробные публичные разборы того, как компания отказалась от дробления и во что оно ей обошлось, существуют — несколько таких случаев с первоисточниками разобраны в третьей части книги. Но их немного, и каждый такой текст становится событием уже потому, что он редкость.

Из этого не следует, что за каждым успешным докладом стоят десятки провалов, о которых молчат. Сколько неудач осталось за кадром, не знает никто, и число здесь было бы догадкой. Редкие публичные разборы доказывают, что такие истории случаются, но ничего не говорят о том, как часто. Утверждать можно только о механизме: выборка, которую видела индустрия, отбирала истории по признаку успеха и масштаба, а у противоположных историй не было ни повода, ни канала выйти наружу. Для ошибки выжившего неважно, много ли было неудач: достаточно того, что их не было видно.

Такая выборка подсказывает вывод, который трудно назвать глупым. Netflix успешен; Netflix построен на микросервисах; значит, микросервисы ведут к успеху. Перед нами были выступления инженеров компании, которая держала огромный трафик, открывала код своих инструментов и подробно рассказывала, как переживает отказы. Чтобы проверить вывод, понадобились бы ещё две группы: компании с другой архитектурой и тем же успехом и компании с той же архитектурой, но без успеха. По той же логике отбора первые редко становились докладами об архитектуре, а вторые почти никогда. При такой выборке силлогизм выглядит не натяжкой, а самым экономным объяснением.

Свою роль сыграл и сам канал. Инженеры слушали инженеров и объясняли успех компании тем, что было видно из зала, — устройством системы. На конференции о маркетинге тот же успех объяснили бы маркетингом, на встрече юристов — лицензионными сделками. Каждая аудитория получала объяснение на своём языке, и нашим языком была архитектура.

Силлогизм принимает совпадение за причину: из того, что две вещи встречаются в одной компании, не следует, что одна порождает другую. Если спросить, что сделало Netflix тем, чем он стал, список получится другим. В нём будут каталог фильмов и сериалов, рекомендательная система, которая подсказывает, что смотреть дальше, и сделки с правообладателями, без которых каталога не было бы. Архитектура позволяла всему этому работать под нагрузкой, но она один фактор из многих. Это вывод, а не измерение: вклад каждого фактора никто не разложил по долям. И всё же подписка оплачивает прежде всего то, что можно посмотреть, а не устройство сервисов по ту сторону экрана.

При этом данные вернувшихся самолётов не ложны. Архитектура Netflix действительно решала задачи Netflix, и доклады о ней рассказывали правду. Благодаря таким выступлениям многие команды впервые увидели в работе *circuit breaker* — механизм, который перестаёт посылать запросы отказавшему сервису, — и проектирование в расчёте на отказ. Слушали их не зря. Ошибка рождается не в наблюдении, а в выводе, который делает слушатель, не видя остальной выборки. И этот механизм не выбирает сторону: он работает и против этой книги, которая тоже собирает удачные истории, только другого рода.

У выборки выживших есть второе искажение — во времени, и его можно назвать ошибкой момента. Доклад показывает компанию в той точке, где её решили показать: после многих лет роста, с сотнями или тысячами инженеров, со своими инструментами против отказов, которые она успела пережить. Схема на экране — снимок конечного состояния. По ней не видно, в каком порядке появлялись её части и что стояло на их месте раньше. К любому докладу об архитектуре можно приложить простую проверку: на каком году жизни компании снята схема и что было на её месте за пять лет до того.

16 января 2007 года Netflix запустил просмотр фильмов через интернет на персональных компьютерах. Сервис предлагал около тысячи фильмов, работал только в Windows и не требовал доплаты от тех, кто уже получал DVD по почте, но ограничивал просмотр месячным лимитом часов, который зависел от тарифа. Основным делом компании оставалась рассылка дисков. Микросервисов у неё не было: описывая в 2016 году завершение переезда в облако, сама компания называет исходную точку монолитным приложением, а конечную — сотнями микросервисов. Переход начался после сбоя базы данных в августе 2008 года и закончился в январе 2016-го, то есть занял около семи лет. Архитектура, которую мы видели на сценах в середине 2010-х, была итогом этих лет, а не их началом. Тем самым Netflix укладывается в наблюдение Фаулера из «MonolithFirst».

В пути есть то, чего нет на итоговой схеме, — причина. У Netflix она стоит в самом начале пути, в августе 2008 года. За следующие годы компания превратилась из рассылки дисков в сервис потокового видео, и вместе с ней менялось то, что требовалось от архитектуры. Повторять путь никто не обязан: ждать собственного сбоя базы данных, чтобы получить право на сервисы, бессмысленно. Но без пути непонятно, какую задачу решает схема, и сверить эту задачу со своей не на чем.

Команда, которая берёт такую схему в самом начале жизни продукта, получает ответ на вопрос, который перед ней ещё не встал и, возможно, не встанет никогда. У неё нет ни проблемы, породившей эту архитектуру, ни лет, за которые её достраивали, ни людей, которые её держат. Две ошибки складываются: выборка показывает только выживших, момент — только их конечное состояние. Копируется пункт назначения, а не путь.

2.3. Книги без контекста

В феврале 2015 года издательство O'Reilly выпустило книгу Сэма Ньюмана «Building Microservices» — «Создание микросервисов». Ньюман работал над ней в 2014 году, когда вышла статья Льюиса и Фаулера, и для многих команд книга стала первым и главным справочником по новому стилю. Устроена книга так, как устроен любой хороший справочник. Первая глава объясняет, что такое микросервисы и чем они полезны. Дальше идут одиннадцать глав, и почти все они о том, как с микросервисами работать: как моделировать сервисы и связывать их между собой, как выделять их из монолита, развёртывать, тестировать, следить за их работой и защищать, что меняется на большом масштабе. Среди этих глав есть и глава о роли архитектора, и глава о законе Конвея и проектировании систем, а последняя сводит сказанное воедино.

О цене Ньюман пишет с первой главы: её последний раздел перед итогами называется «No Silver Bullet» — «Серебряной пули нет». Микросервисы, предупреждает Ньюман, не бесплатный обед и не универсальное средство: они приносят с собой всю сложность распределённых систем. Каждая компания, организация и система устроены по-своему, и подойдут ли им микросервисы, зависит от многих факторов. К этому Ньюман возвращается в последней главе, в разделе «When Shouldn't You Use Microservices?» — «Когда микросервисы не нужны»: для нового продукта он советует сначала рассмотреть монолит. Упрёк «книга не предупреждала» опровергается и первой её главой, и последней.

Со временем Ньюман писал об этом всё прямее. В ноябре 2019 года вышла его книга «Monolith to Microservices» — «От монолита к микросервисам», о переходе от одной архитектуры к другой. В ней он настаивал, что микросервисы — не цель и что само их наличие ничего не выигрывает. Переходить к ним стоит осознанно, ради того, чего нельзя добиться с нынешней архитектурой. В марте 2020 года на конференции QCon в Лондоне он, в передаче журналиста The Register, говорил, что микросервисы — не выбор по умолчанию и что половине своих клиентов он советовал обойтись без них.

Во втором издании «Building Microservices», вышедшем в августе 2021 года, позиция записана без смягчений. Уже в предисловии Ньюман замечает, что для многих микросервисы стали архитектурным выбором по умолчанию и что оправдать это трудно. В первой главе монолит назван вполне законным выбором, а затем — «на мой взгляд, разумным выбором по умолчанию» (the sensible default choice). Сам Ньюман, по его словам, ищет причину, которая убедила бы его взять микросервисы, а не причину от них отказаться. Там же есть раздел о тех, кому микросервисы могут не подойти (Whom They Might Not Work For): для новых продуктов и стартапов они часто плохой выбор, и чем меньше команда, тем заметнее цена. Эту цену, замечает Ньюман, называют налогом на микросервисы (microservice tax), и он прикладывает её к команде из пяти человек: если один из пяти занят этой инфраструктурой, много ценного времени уходит не на продукт.

Значит, искать потерю в тексте бесполезно: уже в первом издании Ньюман записал цену там, где её нельзя не заметить, если читать книгу подряд. Потеря произошла в том, как такую книгу читают.

Раздел первой главы о ключевых преимуществах разбит на семь подразделов, от разнообразия технологий до лёгкости замены. Раздел о том, что серебряной пули нет, один, и стоит он в конце главы. За ним одиннадцать глав, почти все — о том, как это делать. О том, когда микросервисы не нужны, говорит раздел в последней из них, после всех глав об устройстве, а до неё справочник дочитывают редко. Ответ на вопрос «зачем и для кого» занимает несколько страниц в начале и в конце книги, ответ на вопрос «как» — всё остальное.

В этой пропорции нет умысла. Книга о том, как строить, не может не состоять в основном из строительства. Справочник, который на каждой странице сомневался бы, стоит ли строить, был бы другой книгой, и практического руководства у отрасли тогда бы не было. Но у пропорции есть смысл, которого автор в неё не вкладывал: объём сообщает, о чём главный разговор. Из книги выносят ощущение, что вопрос в том, как, а не в том, стоит ли.

К тому же справочник редко читают подряд. Его открывают на главе, которая нужна сегодня: как разделить общую базу, как тестировать взаимодействие сервисов, как развернуть новую версию, не остановив соседей. Первую главу читают один раз, часто бегло, как введение, а к главе о развёртывании возвращаются десятки раз, и в памяти остаётся именно она. Никто не решает пропустить страницы о цене: так читают любой справочник, к которому приходят с задачей.

И приходят к нему, как правило, с уже принятым решением. Книгу о том, как строить микросервисы, открывает команда, которая решила их строить, а решение созрело раньше — на схемах крупных компаний, в разговорах коллег, в том, что считалось современным. Вопрос «стоит ли» к этому моменту кажется закрытым, и страницы, которые на него отвечают, читаются как вступление к главному. Раздел о цене в книге есть, но решение, принятое до того, как её открыли, он уже не меняет.

Второе издание можно прочитать как ответ на такое чтение: разговор о выборе переехал из последней главы в первую. Но вышло оно через шесть с половиной лет после первого, когда положение, описанное в его предисловии, уже сложилось. И само издание осталось руководством о том, как строить: разделы о выборе занимают часть первой главы, а за ней снова идут

главы об устройстве. Новое издание исправило то, что написано, а справочник по-прежнему открывают на главе, которая нужна сегодня.

Ещё сильнее тот же эффект у каталогов паттернов. С 2014 года Крис Ричардсон ведёт сайт microservices.io, где собирает то, что сам называет языком паттернов. В октябре 2018 года в издательстве Manning вышла его книга «Microservices Patterns» — «Паттерны микросервисов» — с 44 паттернами. Паттерн здесь — описанный приём проектирования, у которого есть имя и диаграмма: отдельная база данных у каждого сервиса, сага для операции, которая затрагивает несколько сервисов, шлюз для внешних клиентов перед всеми сервисами. Имя делает приём предметом разговора: на обсуждении архитектуры достаточно сказать «база на сервис», и все понимают, о чём речь. Диаграмма превращает его в чертёж, который можно перенести в свою систему. Приём с именем и чертежом приобретает вес стандарта: раз его внесли в каталог и нарисовали, значит, так принято.

Сам каталог стандартом себя не объявляет. Паттерны на microservices.io описаны по общей форме: контекст, задача, силы, которые приходится уравнивать, решение и то, что получится в итоге, с достоинствами и недостатками. У отдельной базы на сервис недостатки названы прямо: трудно проводить бизнес-операции, которые охватывают несколько сервисов, трудно строить запросы по данным из разных баз, сложно управлять множеством баз. В том же каталоге паттерном описана и монолитная архитектура. Взаимодействия в ней проще понять и отладить, вызовы локальны, операции обычно укладываются в транзакцию одной базы. Там же сказано, что недостатки монолита становятся серьёзнее, когда растут приложение и число команд, которые его разрабатывают.

Но в разговор об архитектуре попадает имя, а не раздел о том, что получится в итоге. В проектном документе остаётся строка «база на сервис», а три её недостатка туда не переходят, как и замечание из описания монолита о числе команд. Имя паттерна занимает два слова, описание его цены — абзац.

Ньюман пишет о команде из пяти человек, каталог связывает недостатки монолита с числом команд. Однако фраза о пяти людях работает, только когда рядом с ней стоит собственное число читателя. Книга же обращается ко всем сразу и не знает, открыли её в компании из пяти инженеров или из пяти тысяч. Сравнить себя с Netflix, чью архитектуру брали за образец, и понять, столько ли у нас инженеров, сколько у него, может только тот, кто держит книгу.

При чтении ради того, как это делать, такому сравнению нет места. А от ответа зависит смысл каждой главы: для организации из пятисот инженеров глава о развёртывании описывает, как командам не ждать друг друга, а для команды из десяти — сколько работы добавится к продукту. Ни одна книга не может спросить своего читателя «Сколько у вас инженеров?». Этот вопрос читатель может задать только себе сам — и именно он выпал.

2.4. Тетралогия хайпа

В августе 2003 года издательство Addison-Wesley выпустило книгу Эрика Эванса «Domain-Driven Design: Tackling Complexity in the Heart of Software» — «Предметно-ориентированное проектирование: как справиться со сложностью в самом сердце программы». Книга была не о сетях. Эванс писал о том, как построить модель сложной предметной области и как разработчикам и знатокам этой области говорить на одном языке. Подход получил короткое имя по заголовку книги — DDD. До статьи Льюиса и Фаулера оставалось десять с половиной лет.

Одно из главных понятий книги — bounded context. В русских переводах его называют ограниченным контекстом, но в речи инженеров прижилось английское название. Мысль за ним простая: у модели есть область, внутри которой она верна, и эту область нужно очертить явно. Эванс предлагал задавать её границы «с точки зрения организации команд, использова-

ния в определённых частях приложения и физического воплощения, такого как кодовые базы и схемы баз данных» (team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas). Команда в этом определении уже есть. О сети и единице развёртывания в нём не сказано ничего.

Легче всего увидеть такую границу на одном слове. Возьмём условную систему гостиницы. Для бронирования «номер» — это категория, вместимость и цена за ночь на выбранные даты. Для хозяйственной службы «номер» — конкретная комната на третьем этаже, которую нужно убрать до двух часов дня. Слово одно, моделей две, и попытка описать обе одним классом даёт объект, наполовину чужой каждой из сторон. Bounded context разрешает словам значить разное по разные стороны границы модели и требует, чтобы место перехода было названо явно.

Через десять с лишним лет это различие встретилось с новым вопросом. Статья 2014 года описывала сервисы, нарезанные по возможностям бизнеса, и тому, кто начинал резать, нужно было знать, где проходит линия. Ответ нашёлся в той же статье: в разделе о данных Льюис и Фаулер ссылались на bounded context Эванса. Они отмечали, что делить предметную область на контексты полезно и для монолитной, и для микросервисной архитектуры, но между границами сервисов и контекстов есть естественное соответствие.

10 июня 2014 года, через три месяца после статьи, Google представил Kubernetes. Версия 1.0 вышла 21 июля 2015 года, через полтора месяца после того, как на сайте Фаулера спорили, с чего начинать новую систему. В тот же день Google объявил о передаче проекта создаваемому фонду Cloud Native Computing Foundation (CNCF). Оркестратор сделал запуск ещё одного сервиса будничным делом: каждая развёрнутая часть системы получала свой контейнер и свой адрес в сети.

Цепочка DDD, bounded context, микросервис, Kubernetes выглядит так, будто каждый шаг продолжает предыдущий. Но на каждом шаге слово «граница» меняет смысл. Bounded context — граница модели и языка: по разные её стороны «номер» значит разное. Граница команды — организационная граница: она отделяет тех, кто договаривается друг с другом каждый день, от тех, с кем договариваются через контракт, и по закону Конвея система стремится её повторить. Микросервис — граница процесса и развёртывания: отдельная программа, которую собирают, выпускают и откатывают сама по себе. Сетевой вызов (network call) — физическая граница, и налог распределённости начисляется именно на неё: на сетевую границу, место, где вызов функции (function call) заменён сетевым вызовом.

Эти четыре границы проводят разные люди и по разным причинам. Границу модели находят в разговорах со знатоками предметной области, когда выясняется, что одно слово у двух отделов значит разное. Границу команды проводит тот, кто решает, сколько людей нанять и кому что поручить. Границу процесса задаёт инженерное решение о том, что выпускается вместе. Сетевую границу создаёт выбор, что будет вызовом функции, а что — пакетом, который уходит по сети. У каждой из четырёх свой вопрос, и ответ на один из них не предreshает ответа на другой.

Поэтому провести их можно в разных местах. Бронирование и хозяйственная служба могут быть двумя модулями одной программы: у каждого своя модель и свой язык, между ними явный интерфейс, а обращение из одного в другой остаётся вызовом функции. Обе модели может вести одна команда из шести человек, и выпускать их она будет вместе. Те же два контекста можно развернуть как два сервиса, и тогда то же обращение пройдёт через сеть. Граница модели в обоих вариантах одна и та же, изменились только граница развёртывания и сетевая. Бывает и наоборот: одна команда держит несколько контекстов, а один контекст занимает несколько сервисов.

Цепочка склеила четыре границы в одну: «один bounded context = одна команда = один deployable = один сетевой адрес», где deployable — единица развёртывания, то, что выпуска-

ется как одно целое. У каждого равенства по отдельности есть основание. Контекст и команду связывает уже определение Эванса; команде, которой нужна независимость, удобно выпускать свой код самой; развёрнутая часть системы в Kubernetes получает адрес по умолчанию. Но первое равенство верно, когда команд много, второе — когда им действительно мешают общие выпуски, а третье описывает свойство инструмента, а не требование задачи. Составленные вместе, они дают равенство, которого не было ни в одном звене: контекст равен сетевому адресу. Логическая граница приравнивается к физической.

С этого момента вопрос о физике системы решается на языке моделирования. Число сетевых границ задаёт уже не число команд, которым нужна независимость, а то, насколько подробно мы разобрались в предметной области. Чем глубже анализ, тем больше контекстов, и каждый новый контекст по формуле означает ещё один сетевой адрес. Хорошее моделирование начинает увеличивать счёт за инфраструктуру, хотя ни книга Эванса, ни статья 2014 года не превращали число контекстов в число сервисов. Именно эту подмену в 2023 году назовут исследователи Google в статье «Towards Modern Development of Cloud Applications» — «К современной разработке облачных приложений». По их словам, микросервисы смешивают логические границы, то есть то, как написан код, с физическими — тем, как он развёрнут.

На практике склейка выглядит буднично. Допустим, систему для той же условной гостиницы пишет команда из шести человек. Она проводит двухдневную сессию моделирования и добросовестно находит на доске восемь контекстов. Работа сделана хорошо: модели разведены, у каждого контекста свой язык. На следующей неделе появляются восемь репозиторийев, восемь конвейеров развёртывания и восемь сетевых адресов, и никто не принимал такого решения отдельно. Оно уже содержалось в формуле, по которой результат моделирования переводится в устройство системы.

В организации из сотен команд такая подмена может пройти почти незаметно: контексты там нередко и так совпадают с командами. В команде из шести человек она определяет устройство всей системы. Вопрос о том, нужны ли шести людям восемь независимых выпусков, на сессии моделирования не звучал: его не было в повестке.

Соответствие сервиса контексту поддерживал и сам автор понятия. В 2015—2016 годах Эванс выступал в Лондоне и Берлине с темой «DDD & Microservices: At Last, Some Boundaries!» — «DDD и микросервисы: наконец-то границы». По отчётам InfoQ, он видел в микросервисах, возможно, лучшую среду для DDD за всю его историю, потому что они дают настоящие границы, а соответствие одного сервиса одному контексту считал хорошим вариантом хотя бы для первого приближения. В 2019 году на DDD Europe он говорил на тему «Language in Context» — «Язык в контексте». Там, в пересказе того же InfoQ, он назвал отождествление микросервиса с bounded context упрощением, а сами микросервисы — одновременно самой большой возможностью и самым большим риском за долгое время. Уточнение шло в другую сторону: контекст может охватывать кластер сервисов, спроектированных вместе, а может не содержать сервисов вовсе, только сообщения, схемы и протоколы. Мысль, что контекст может быть меньше процесса и жить модулем рядом с другими, — уже позиция этой книги.

Если довести формулу до конца, получится крайний случай, который называют наносервисами. Когда контексты мельчают, линия разреза перестаёт идти по модели и начинает идти по таблице базы данных: сервис на сущность или на функцию. В гостиничной системе это сервис «номер», сервис «гость», сервис «бронь» и сервис «тариф», у каждого своя таблица. Злоупотребления формулой здесь нет — это её логический предел.

Предел выглядел привлекательно по понятной причине. Если граница сервиса совпадает с границей модели, то маленький сервис кажется признаком тщательного моделирования, а размер становится мерой качества нарезки. Тот, кто выделял сервис на каждую таблицу, не нарушал правило, а исполнял его последовательнее других. При такой нарезке исчезает и сама

граница модели. «Номер» бронирования и «номер» хозяйственной службы снова становятся одной записью, только теперь по ту сторону сетевого вызова.

Отсюда не следует, что ошибся DDD. Это ценный инструмент моделирования, и его ценность от этой ошибки не зависит: общий язык со знатоками предметной области и явные границы моделей нужны и монолиту. Контексты прекрасно живут модулями внутри одного процесса, каждый со своей моделью, своим интерфейсом и при необходимости своей схемой в общей базе. Такой модуль с явным интерфейсом не закрывает дорогу к сервису: если у контекста однажды появится своя команда, его можно будет вынести за сетевую границу.

У сетевой границы есть сильный довод: её нельзя пересечь незаметно, а границу модуля держат только дисциплина команды и проверка зависимостей при сборке. По сути это довод Тилкова о том, что части монолита неизбежно срastaются, и он реален. Но он о том, чем удобнее охранять границу модели, а не о том, где она проходит. Kubernetes здесь тоже ни при чём: монолит в нём живёт не хуже, и оркестратор не требует дробить приложение.

Четыре границы могут совпасть, и иногда это верное решение. Там, где работают сотни команд и каждой нужна независимость, граница модели, граница команды, граница процесса и сетевая граница нередко проходят в одном месте. Но такое совпадение следует из числа команд; определение bounded context его не требует. Ошибка была не в DDD и не в сервисах, а в знаке равенства между bounded context и сетевым адресом.

2.5. Как исчезают оговорки

Статья Льюиса и Фаулера в нынешней редакции на сайте Фаулера занимает больше шести тысяч слов — это около получаса чтения. Многие оговорки в ней стоят в тех же абзацах, что и выводы, а главная — что судить о будущем стиля рано — отдана заключительному разделу. Кто дочитал статью до конца, знал, при каких обстоятельствах получены её выводы. Заметки, вышедшие следом, добавили к описанию ещё три обстоятельства — предпосылки, цену и очерёдность. Вместе эти тексты и есть первое звено цепочки, по которой описание дошло до команд, никогда не открывавших исходного текста. Звеньев в ней по меньшей мере семь, и на каждом текст становится короче.

Второе звено — доклад. Докладчик рассказывает о системе своей компании и рассказывает правду: как делили, что ломалось, какие инструменты пришлось написать самим. Число инженеров, зрелость платформы, годы жизни системы до разделения для него не оговорка, а обстановка. Опускает он то, что считает само собой разумеющимся. Для слушателя из другой компании именно эта обстановка и была условием.

Вернуть его у доклада есть одна возможность — вопросы из зала. Но в оставшиеся несколько минут спрашивают о том, ради чего пришли: каким инструментом делили данные, как переключали трафик, как обходились без общих транзакций. Вопрос о численности компании звучит как вопрос не по теме. Слушатель пришёл узнать, как это делается, а не выяснять, похож ли он на того, кто стоит на сцене. Даже если такой вопрос прозвучит, ответ на него останется в зале: в пересказ выступления попадёт только схема.

Третье звено — пост в блоге с заголовком вроде «10 преимуществ микросервисов». Такой текст обычно пишет тот, кто слышал несколько выступлений и читал статью, и пишет добросовестно: каждый пункт можно найти у источников. Типичные пункты — независимое развёртывание, масштабирование отдельных частей, свобода выбора технологий, изоляция отказов. У перечня преимуществ нет строки для обстоятельств: пункт «при тысяче инженеров» не похож на преимущество и в список не попадает. Ссылки в таком посте ведут на выступления и на другие посты, то есть на предыдущее звено, а не на первое. Следующий автор, который возьмёт этот пост за источник, получит перечень, у которого уже нет происхождения.

Четвёртое звено — обзорный документ поставщика инфраструктуры, то, что по-английски называют *whitpaper*. Он опирается уже не на опыт одной компании, а на практику отрасли, и типичная его фраза звучит как «ведущие компании перешли на микросервисы». Конкретная организация со своим числом инженеров становится «ведущими компаниями», а у них нет численности, с которой можно сверить свою. Истории отдельных компаний в нём сжаты до коротких справок с названием и результатом: перешли, ускорили выпуск, выдержали пиковую нагрузку. Сколько лет занял переход и сколько людей его делали, в такую справку не помещается. Решение о переходе в самом документе принято до первой страницы: дальше описаны этапы пути, а вопрос о его начале вынесен за скобки.

В объявлении о вакансии, пятом звене, от тезиса остаётся одно имя — строка «опыт микросервисной архитектуры». В строке нет ни компании, ни масштаба, ни глагола. Архитектура, которую несколько организаций выбрали для себя, становится навыком, который человек переносит из компании в компанию. Список требований устроен как список преимуществ: место для умения в нём есть, а для обстоятельств, в которых это умение нужно, нет.

Шестое звено — собеседование по проектированию систем (*system design*). Кандидату предлагают за час спроектировать что-нибудь знакомое, например ленту новостей или сервис коротких ссылок. У хорошего ответа есть узнаваемый шаблон: шлюз на входе, сервисы по сущностям, у каждого своя база, очередь между ними. В типичной формулировке задачи задана нагрузка — миллионы пользователей, тысячи запросов в секунду, — а размера команды, которая будет систему писать и сопровождать, в ней нет. Из двух величин, от которых зависит ответ, в условии остаётся одна: про машины, а не про людей. Сам шаблон передаётся так же, как всё в этой цепочке: разборы типовых задач пересказывают удачные ответы, а удачные ответы пересказывают выступления крупных компаний.

Седьмое звено — архитектура стартапа из восьми человек. Её рисует инженер, который читал такие посты и проходил такие собеседования, и рисует так, как его учили отвечать у доски. На такой схеме легко может оказаться больше сервисов, чем людей в команде. Из трёх умений, которые Фаулер в 2014 году называл предпосылками, у восьми человек может не оказаться ни одного. Ни на одном звене вопрос о том, для кого всё это окупается, не был опровергнут. Он перестал звучать между первым звеном и вторым, и дальше его не было в тексте, который передавали.

Налог распределённости при этом начисляется полностью: его размер не зависит от того, дошло ли условие до того, кто платит. А дохода, ради которого налог когда-то платили, — независимости многих команд друг от друга — у восьми человек нет, потому что команда у них одна.

Цепочка работала не один год. Если судить по Google Trends, интерес к слову «микросервисы» рос с 2014 года и продолжал расти до конца десятилетия. Это иллюстрация, а не доказательство: кривая поисковых запросов показывает любопытство, а не число систем, построенных по рецепту, и не говорит, где искавшие встретили слово. Можно предположить, что большинство встретило его после статьи 2014 года, на одном из следующих звеньев. Каждый новый читатель входил в цепочку ниже предыдущего и получал текст ещё короче.

Вывод и условие переносят пересказ по-разному. Вывод говорит о предмете — о сервисах, базах, развёртывании — и остаётся верным, кто бы его ни повторял. Условие говорит о том, кто говорит: «у нас тысяча инженеров» верно только в устах докладчика, и уже следующий пересказчик не может сказать «у нас». Ему остаётся назвать чужое число или опустить его, и опустить проще. Поэтому с каждым звеном из текста уходит то, что держалось на положении предыдущего рассказчика: его численность, его платформа, годы его системы.

Есть и вторая причина, связанная с адресатом. От звена к звену аудитория шире: от зала, который слушает доклад, до каждого, кто готовится к собеседованию. Оговорка о том, для каких организаций это верно, делит такую аудиторию на тех, к кому текст относится, и тех, к

кому нет. Для большинства читателей она означала бы, что текст не про них. Без неё сказанное годится для любого читателя и именно в таком виде расходится дальше.

Сокращение само по себе не порок. Эта цепочка — обобщение, а не история одного текста: звенья могли меняться местами или выпадать, но направление у неё одно, от описания к требованию. Без пересказа знание не выходит за пределы одной организации, и та же цепочка разнесла по отрасли то, что пригодилось почти всем: автоматизацию развёртывания, мониторинг, привычку проектировать в расчёте на отказ. Краткость и не обязана губить оговорку. Мысль Ньюмана о том, что чем меньше команда, тем заметнее цена микросервисов, укладывается в несколько слов и вся состоит из условия. Значит, дело не в длине текста, а в том, какую часть фразы при сокращении считают необязательной.

В статье тезис о микросервисах звучал так: опыт положительный, но судить рано, а в заметке о премии добавлялось, что, если система не слишком сложна для монолита, микросервисы не стоит и рассматривать. В докладе от него остаётся «мы перешли на микросервисы, и наши команды выпускают изменения, не дожидаясь друг друга». Это правда, и в ней ещё есть «мы» и «наши команды», за которыми стоит тысяча инженеров, но число не произнесено. Фразы из статьи и из доклада друг другу не противоречат, и каждый из авторов мог бы подписаться под обеими.

Условия первого звена, сведённые в одну фразу, звучали бы примерно так: «если система сложна, платформа зрела, а инженеров тысячи, микросервисы окупаются». Целиком эту фразу никто не произносил: в статье и заметках её первая половина выражалась через сложность задачи и умения команды, в докладе стояла за словами «мы» и «наши команды». До вакансии дошла только вторая половина, и от неё осталось название навыка — «опыт микросервисной архитектуры». На каждом шаге теряются условия и сохраняются выводы. Никто не лжёт: каждый честно пересказывает предыдущий шаг, просто короче.

Рецепт, которым закончилась эта история, — не искажённый текст, а верный текст без условия. Каждый его вывод можно найти в первоисточнике, и там он верен — при обстоятельствах, которые первоисточник называет. Выпало одно «если» — та часть фразы, которая говорила, для кого выводы сделаны. Поэтому опровергать в рецепте нечего, а искать тех, кто его искажил, бесполезно: их не было. Вернуть можно только выпавшую переменную — число инженеров. Стоит поставить рядом с любым выводом рецепта численность своей организации, и он снова становится утверждением, которое можно проверить.

Летом 2015 года эта оговорка лежала на виду: о том, с чего начинать и во что обходится распределённость, два инженера спорили на сайте самого Фаулера, открыто, под своими именами. В 45-минутный доклад она не поместилась: её никто не вычёркивал, для неё просто не нашлось минуты. Не дошла она и дальше — ни до строки в объявлении о вакансии, ни до схемы, которую рисует для своей системы стартап из восьми человек. У этих восьми есть все выводы, сделанные когда-то о компаниях с тысячами инженеров, и нет числа, ради которого их делали.

Рецепт существует как текст без автора и без даты: его передают дальше, не называя источника. Цепочка честного пересказа объясняет, как из него выпало условие. Она не объясняет, почему условие никто не вернул, хотя исходные тексты все эти годы оставались на месте и подняться к ним по цепочке мог любой. Пересказ потерянного не возвращает: возвращают те, кому это нужно. А рецепт без условия продержался годы. Сами по себе рецепты так долго не живут — долго живут только те, что кому-то выгодны.

Глава 3. Почему это прижилось

В 2020 году исследователи из Штутгартского университета опросили 591 человека о том, как нанимают разработчиков. Девяносто семь из них ответили дважды: как те, кто нанимает, и как те, кого нанимают. Один и тот же человек пишет текст вакансии и обновляет собственное резюме. В системе, которая сделала рецепт нормой, нет наблюдателей со стороны.

Устойчивость рецепта проще всего объяснить чьей-то ошибкой суждения. Точнее другое объяснение: каждый участник вёл себя разумно на своём месте, и из суммы таких решений сложился результат, которого никто не выбирал. Ошибку закрепили не отдельные инженеры: её закрепила система, в которой они работали.

Инженеру сложный проект приносил строку в резюме и довод при повышении. Руководителю следование норме давало защиту при любом исходе, и слово «монолит» к тому времени звучало как оценка, а не как описание. Рынку каждый новый сервис приносил выручку, причём выручку за решения реальных проблем. К концу десятилетия появились первые публичные отказы, но спор о них так и остался спором предпочтений.

У простого решения на всех этих уровнях одно общее свойство: его не видно. Сложное оставляет следы на каждом шагу: диаграммы, доклады, вакансии, счета. Простое не оставляет ни строки в резюме, ни пункта в годовом отчёте, ни позиции в чьём-либо счёте.

3.1. Карьерные стимулы

Две строки из двух резюме. В первой перечислено: микросервисы, Kubernetes, Kafka, Istio. Во второй сказано скромнее: монолит на Rails, один сервер. Первая строка совпадает с ключевыми словами вакансии и проходит первичный отбор. Вторая не совпадает ни с одним из них, хотя за ней может стоять система, которая годами приносит компании выручку, выдерживает сезонные пики и не требует ночных дежурств.

Автор второй строки может знать о нагрузке и отказах не меньше, чем автор первой. Разница в том, что его знание ушло в решение не усложнять: он понимает, почему в его системе не понадобились очереди и почему одной базы хватает. Это знание нельзя перечислить через запятую. Первичный отбор ведёт человек, который сверяет резюме со списком из вакансии, или программа, которая делает то же самое быстрее. Ни тот ни другая этой разницы не видят: они сравнивают слова, а не системы.

Инженер, который читает вакансии, делает из этого разумный вывод. Следующий проект стоит выбирать так, чтобы через пару лет в резюме оказалась первая строка. У такого поведения есть имя в литературе — *resume-driven development*, разработка ради резюме: выбор технологий по их ценности для резюме, а не для продукта. Название звучит как упрёк, будто инженер жертвует продуктом ради собственной карьеры. Эмпирическое исследование явления рисует картину, в которой упрёк адресовать некому.

Опрос 2020 года провели Йонас Фрич, Марвин Вирих, Юстус Богнер и Штефан Вагнер; результаты вышли в 2021 году. Среди участников 130 ответили как специалисты по найму, 558 — как разработчики и другие технические специалисты. Почти 60% нанимающих согласились, что на тексты их вакансий влияет то, какие технологии сейчас на подъёме. Но когда их спросили, кого из двух кандидатов они предпочтут, опыт с новейшими популярными технологиями выбрали только 20%, а опыт с устоявшимися — около 39%. Знание устоявшихся технологий назвали важным около 85% нанимающих, популярных — около 59%.

Разработчики видят ту же картину с другой стороны. Из них 82% согласились, что работа с популярными технологиями делает их привлекательнее для будущих работодателей. Около

63% считали, что чем больше таких технологий они освоят, тем привлекательнее станут. Каждый пятый подтвердил, что его команда хотя бы раз брала популярную технологию, зная, что для задачи она не лучший выбор.

Главный результат исследования — расхождение. Нанимающие пишут популярные технологии в вакансии, но при выборе чаще ценят устоявшиеся, а девять из них в свободных ответах назвали третье: готовность учиться, умение работать с людьми, понимание фундаментальных принципов. Разработчики выносят популярные технологии в резюме, потому что считают, что их оценивают именно по ним. Авторы определяют явление как взаимодействие специалистов по найму и разработчиков, при котором и вакансии, и резюме перегружены популярными технологиями, хотя обе стороны на деле ценят опыт с ними меньше. Среди возможных последствий они называют снижение качества программ и текучку, которая возникает из ложных ожиданий обеих сторон.

Каждая сторона здесь отвечает на то, что показывает другая, а не на то, что та ценит. Авторы осторожно добавляют, что у такого взаимодействия есть потенциал стать самоподдерживающимся: соискатели могут откликаться на акцент вакансий ещё сильнее, выдвигая те же технологии в резюме, и тем самым его подкреплять. Сигнал держится на самом устройстве обмена, и поэтому переживает любое отдельное собеседование и любого отдельного интервьюера. Тот, кто хотя бы раз поднимал Kubernetes повыше в списке навыков, делал ровно то, чего от него, по всем видимым признакам, ждали. Тщеславия в этом нет: это верный ответ на вопрос, который задаёт рынок, пусть сам вопрос задан неточно.

У этих чисел есть границы. Выборка самоотобранная: участников набирали в мае—июле 2020 года через немецкий журнал для разработчиков, профессиональные и социальные сети, около 90% из них — из Германии. Зато опрос прошёл в самом конце подъёма микросервисов и даёт снимок рынка труда по свежим следам. Он касается технологий вообще и отдельно микросервисы не измеряет, хотя в определении популярной технологии, которое видели участники, они стоят первым примером. Большая часть чисел описывает убеждения, а не поступки: о том, что их команда сознательно брала не лучшую для задачи технологию, сообщила примерно пятая часть, то есть это поведение меньшинства. Но стимулу не нужно большинство: достаточно, чтобы в спорных случаях, когда защитимы оба варианта, выбор раз за разом смещался в одну сторону.

Внутри компании действует та же логика, только вместо вакансии — оценка работы. Строка «мигрировал на микросервисы» описывает проект с началом, концом и диаграммой: было одно приложение, стало двенадцать сервисов, вот схема до, вот схема после. У проекта есть сроки, команда, документ с обоснованием и день, когда последний модуль переехал. Строка «выбрал простое решение, и оно три года работает без инцидентов» описывает то, чего не произошло. У предотвращённой проблемы нет презентации: нечего вывести на слайд, некого поблагодарить в рассылке, нет даты, которую можно отметить.

Дело не в том, что руководители любят сложность. Любая процедура, которая оценивает работу по предъявленным результатам, вознаграждает то, что можно предъявить. Миграция производит предъявляемое в избытке. Отсутствие инцидентов не производит ничего, и со стороны его легко принять за везение или за простоту задачи. Руководитель, который оценивает, сам отчитывается выше, и ему тоже нужно показать, что его команда сделала за год. Размах проекта к тому же читается как размах ответственности, а ответственность — привычный довод в разговоре о повышении.

Невидимость легко проверить на своей компании. Достаточно найти в её документах хотя бы одно решение не делать чего-то сложного, за которое автор получил признание. Руководитель группы, который за квартал убедил команду не выделять платёжный модуль в отдельный сервис, сделал работу: собрал доводы, посчитал нагрузку, выдержал спор. Результат этой

работы — сервис, которого нет. В форме самооценки для него нет поля, а в рассказе о нём нет ни одной диаграммы.

В этой асимметрии есть и сдвиг во времени. Налог за каждую новую сетевую границу компании платит годами: в эксплуатации, в дежурствах, в разборах отказов. Награду за её создание автор получает один раз и сразу, при ближайшей оценке. Ни резюме, ни документ на повышение не показывают, окупилась ли граница. Когда приходит счёт, автор решения может работать уже над следующим проектом или в другой компании.

Бывают миграции, которые действительно нужны, и тогда награда заслужена. Сотни инженеров, которые блокируют друг друга в одном приложении, — реальная проблема, и тот, кто её решил, сделал для компании много. Слабость процедуры оценки в другом: у неё нет способа отличить нужную миграцию от ненужной, потому что в отчёте о сделанном обе выглядят одинаково. Автора простого решения никто не наказывает. Его просто не видно.

На входе в компанию действует третий механизм — собеседование по проектированию систем (system design). Кандидату дают задачу вроде сервиса коротких ссылок или ленты новостей и час у доски. Шаблон ответа известен заранее: балансировщик, кеш, очередь, несколько сервисов, реплики базы. Для сервиса коротких ссылок на доске появляются парк серверов приложений, кеш перед хранилищем, очередь для сбора статистики и хранилище ключей, распределённое по нескольким машинам, — там, где работу могла бы сделать одна таблица с индексом по короткому ключу. Интервью проверяет знание распределённых паттернов и поэтому вознаграждает их применение. Чем больше приёмов кандидат уместно назвал, тем больше он показал.

Ответ «монолит на PostgreSQL» для большинства компаний правильный. На собеседовании он проигрывает, потому что не демонстрирует тех знаний, ради проверки которых собеседование устроено. Кандидат, который скажет, что при заявленной нагрузке хватит одного сервера и хорошо настроенной базы, даст верный ответ и оставит интервьюера без материала для оценки. Естественное продолжение такого разговора — вопрос, что будет при нагрузке в сто раз больше: формат ищет место, где понадобятся паттерны, и находит его. Кандидаты быстро это понимают и учатся рисовать распределённую схему уверенно и подробно.

Вины интервьюера здесь тоже нет. Инженеру, который будет проектировать системы, знание распределённых паттернов нужно, и за час убедиться в нём иначе трудно. Ограничение заложено в самом формате: он хорошо проверяет, знает ли кандидат приёмы, и плохо проверяет суждение о том, когда они не нужны. Кроме того, формат легко воспроизводит себя: интервьюеры сами когда-то проходили такие собеседования и знают, как выглядит хороший ответ. Суждение «здесь это лишнее» трудно показать на доске. У него нет схемы.

Резюме, оценка и собеседование устроены по-разному, но поощряют одно и то же. Разработчик, который выносит Kafka в первую строку, верно читает вакансии. Руководитель, который вписывает Kafka в вакансию, отражает рынок; комиссия, которая повышает автора миграции, оценивает то, что ей показали; интервьюер проверяет то, что умеет проверить. Ни один из этих шагов не требует ни тщеславия, ни глупости. И всё же их сумма складывается в архитектуры, которые вряд ли выбрал бы человек, отвечающий за компанию целиком и на годы вперёд.

Выйти из этой системы в одиночку дорого, и в этом её устойчивость. Тот, кто выбирает простое, платит сам: строкой в резюме, доводом при повышении, баллами на собеседовании. Выгоду от его выбора получает компания, причём незаметно для всех, включая её саму. Тот, кто выбирает сложное, получает награду лично, а расплачивается за его выбор вся компания — вразбивку, по разным бюджетам, так что ни в одной отдельной статье расходов эта плата не видна. При таком раскладе простое решение обходится инженеру личной жертвой, и требовать её от каждого было бы странно.

Эту систему не придумал ни один человек. Её собрали мы — те, кто писал вакансии, сидел в комиссиях, проводил собеседования и обновлял собственные резюме. Индивидуально

рациональное поведение даёт коллективно иррациональный результат. Индустрия создала систему, которая вознаграждает сложность.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.