ГЛАВА 1

Что такое программная инженерия?

Автор: Титус Винтерс
Редактор: Том Маншрек

> Ничто не строится на камнях; все построено на песке, но мы должны строить так, как если бы песок был камнем.
>
> *Хорхе Луис Борхес*

Программирование и программная инженерия имеют три важных отличия: время, масштаб и компромиссы. В отличие от обычных программистов, инженеры-программисты должны больше внимания уделять течению времени и необходимости внесения изменений, больше заботиться о масштабе и эффективности как самого ПО, так и организации, которая его производит. Инженеры-программисты принимают более сложные решения, дороже платят за ошибки и часто опираются на неточные оценки времени и роста.

Мы в Google иногда говорим: «Программная инженерия — это программирование, интегрированное во времени». Программирование, безусловно, является важной частью программной инженерии: в конце концов, именно в процессе программирования создается новый софт. Но если мы разделяем понятия, то нужно разграничить задачи программирования (разработку) и программной инженерии (разработку, изменение, сопровождение). Время — это новое важное измерение в программировании. Куб — это не квадрат, расстояние — это не скорость. Программная инженерия — это не программирование.

Чтобы понять, как время влияет на программу, задумайтесь: «Как долго будет жить[1] код?» Крайние оценки в ответах на этот вопрос могут отличаться в 100 000 раз. Легко представить код, который просуществует несколько минут, или другой код, служащий десятилетиями. Как правило, короткоживущий код не зависит от времени: едва ли нужно адаптировать утилиту, которая проживет час, к новым версиям базовых библиотек, операционной системы (ОС), аппаратного обеспечения или языков

[1] Мы не имеем в виду «продолжительность выполнения», мы имеем в виду «продолжительность поддержки» — как долго код будет продолжать развиваться, использоваться и поддерживаться? Как долго это ПО будет иметь ценность?

программирования. Недолговечные системы фактически являются «мимолетными» задачами программирования и напоминают куб, который сжат вдоль одного измерения до вида квадрата. Но с увеличением продолжительности жизни кода изменения становятся более важными для него. За десяток лет большинство программных зависимостей, явных или неявных, почти наверняка изменятся. Понимание этого явления помогает отличать программную инженерию от программирования.

Это отличие лежит в основе *устойчивости* в мире ПО. Проект будет более *устойчив*, если в течение ожидаемого срока его службы инженер-программист сможет сохранить способность реагировать на любые важные технические или коммерческие изменения. Именно «сможет» реагировать в том случае, если обновление будет иметь ценность[1]. Когда вы теряете способность реагировать на изменения в базовых технологиях или развитии продукта, не надейтесь, что такие изменения никогда не превысят критического уровня. В условиях проекта, развивающегося несколько десятилетий, такие надежды обречены на провал[2].

Взглянуть на программную инженерию можно со стороны оценки масштаба проекта. Сколько людей вовлечено в разработку? Как меняются их роли при разработке и сопровождении проекта с течением времени? Программирование часто является актом индивидуального творчества, но программная инженерия — это командная работа. Одно из первых и самых точных определений программной инженерии звучит так: «Разработка многоверсионных программ для большого числа людей»[3]. Оно означает, что различие между программной инженерией и программированием определяется количеством пользователей и сроком действия продукта. Командная работа создает новые проблемы, но также открывает такие возможности для создания систем, какие не может предложить один программист.

Организация команды, состав проекта, а также стратегия и тактика его развития — важные компоненты программной инженерии, которые зависят от масштаба организации. Растет ли эффективность производства софта по мере увеличения организации и расширения ее проектов? Растет ли эффективность рабочего процесса по мере развития организации и насколько пропорционально этому растет стоимость стратегий тестирования и управления версиями? Проблемы масштаба, связанные с увеличением числа сотрудников и налаживанием коммуникации между ними, обсуждались с первых дней программной инженерии, начиная с появления

---

[1] Это вполне точное определение технического долга: он возникает тогда, когда что-то «должно» быть сделано, но еще не сделано, и является разницей между текущим кодом и будущим желаемым кодом.

[2] Тут вполне уместен вопрос: как заранее узнать, что проект будет долгосрочным?

[3] Сейчас трудно установить авторство определения: одни считают, что впервые оно было сформулировано Брайаном Рэнделлом (Brian Randell) или Маргарет Гамильтон (Margaret Hamilton), другие — что автором является Дейв Парнас (Dave Parnas). Это определение часто приводят как цитату из отчета «Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee», Рим, Италия, 27–31 октября 1969 г., Брюссель, отдел по научным вопросам, НАТО.

мифического человеко-месяца[1]. Часто они имеют политический характер, и от их решения во многом зависят устойчивость ПО и ответ на вопрос: «Как дорого обойдется то, что придется делать снова и снова?»

Еще одно отличие программной инженерии от программирования заключается в сложности принятия решений и цене ошибок. Постоянная оценка компромиссов между несколькими путями движения вперед часто основана на несовершенных данных, и иногда цена ошибки очень высока. Работа инженера-программиста или лидера команды инженеров состоит в стремлении к устойчивости организации, продукта и процесса разработки, а также в управлении ростом затрат. Иногда можно откладывать технические изменения или даже принимать плохо масштабируемые политики, которые потом придется пересмотреть. Но, делая такой выбор, инженер-программист должен понимать, к каким затратам эти решения приведут в будущем.

Универсальные решения в программной инженерии встречаются редко, как вы увидите и в этой книге. Учитывая разброс в 100 000 раз между ответами на вопрос «Как долго будет жить ПО?» и в 10 000 раз — между ответами на вопросы «Сколько инженеров работает в компании?» и «Сколько вычислительных ресурсов доступно для проекта?», опыт Google почти наверняка не будет соответствовать вашей ситуации. Поэтому в этой книге мы постарались показать, как в Google искали правильные пути в разработке и сопровождении ПО, рассчитанного на десятилетия, имея десятки тысяч инженеров и вычислительные ресурсы мирового масштаба. Большинство методов, применение которых мы считаем обязательным в таком масштабе, также хорошо подойдут для меньших организаций: считайте эту книгу отчетом одной инженерной экосистемы, который может вам пригодиться. Иногда сверхбольшие масштабы связаны с повышенными расходами, и, возможно, благодаря нашим предупреждениям, когда ваша организация вырастет до таких масштабов, вы сможете найти более удачное решение по издержкам.

Прежде чем перейти к обсуждению особенностей командной работы, культуры, политики и инструментов, давайте подробнее рассмотрим время, масштаб и компромиссы.

Время и изменения

Когда человек учится программировать, срок действия его кодов обычно измеряется часами или днями. Задачи и упражнения по программированию, как правило, пишутся с нуля, почти без рефакторинга и, конечно, без долгосрочного сопровождения. Часто после первого использования эти программы не пересобираются и не применяются на практике. Получая среднее или высшее образование, вы могли участвовать в групповой реализации проекта, который живет месяц или чуть дольше. Такой проект может включать рефакторинг кода, например в ответ на из-

[1] *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. СПб.: Питер, 2021. — *Примеч. пер.*

меняющиеся требования, но маловероятно, что он будет зависеть от более широких изменений в его окружении.

Разработчиков короткоживущего кода можно также найти в некоторых отраслях. Мобильные приложения, например, обычно имеют довольно короткий срок действия[1] и часто переписываются заново. На ранних стадиях развития стартапов инженеры могут вполне обоснованно сосредоточиться на ближних целях в ущерб долгосрочным инвестициям, поскольку сама компания, в которой они работают, может просуществовать недолго. Разработчик стартапа вполне может иметь десятилетний опыт разработки и почти или совсем не иметь опыта поддержки ПО, которое должно работать дольше одного-двух лет.

С другой стороны, есть успешные проекты с практически неограниченным сроком службы: трудно предсказать, когда прекратит свое существование Google Search, ядро Linux или Apache HTTP Server. Большинство проектов Google должны существовать неопределенно долго и периодически претерпевать обновления зависимостей, языковых версий и т. д. С течением времени такие долгоживущие проекты *рано или поздно* начинают восприниматься иначе, чем задачи по программированию или развитию стартапа.

На рис. 1.1 показаны два программных проекта на противоположных концах спектра «ожидаемого срока службы». Как обслуживать проект с ожидаемым сроком службы, измеряемым часами? Должен ли программист бросить все и заняться обновлением, если во время работы над сценарием на Python, который будет выполнен всего один раз, вышла новая версия ОС? Конечно, нет: такое обновление некритично. Но если проект Google Search, находящийся на противоположном конце спектра, застрянет на версии ОС 1990-х годов, обслуживание станет проблемой.

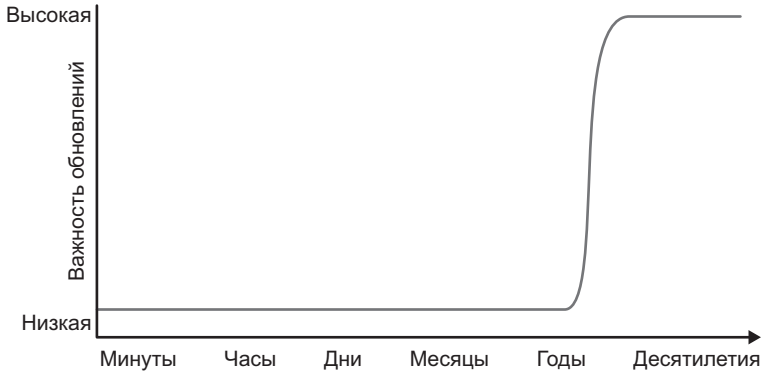

Рис. 1.1. Срок жизни и важность обновлений

[1] Как заявляют в компании Appcelerator, «ничто в мире не определено, кроме смерти, налогов и короткого срока службы мобильных приложений» (https://oreil.ly/pnT2_, блог Axway Developer, 6 декабря 2012 года).

Наличие точек на спектре сроков службы, соответствующих низкой и высокой важности обновлений, предполагает, что где-то есть переход. Где-то на линии, соединяющей одноразовую программу и проект, развивающийся десятилетиями, есть этап появления реакции проекта на изменение внешних факторов[1]. Любой проект, в котором изначально не планировались обновления, переживает переход болезненно по трем причинам, каждая из которых усугубляет две другие:

- обновления еще не выполнялись в этом проекте: по ним есть только предположения;
- инженеры едва ли имеют опыт проведения обновлений;
- большой объем обновлений: одномоментно приходится применять обновления, накопившиеся за несколько лет, вместо постепенного применения небольших обновлений.

Далее, выполнив такое обновление один раз (полностью или частично), вы рискуете переоценить стоимость следующего обновления и решить: «Никогда больше». Компании, которые приходят к такому выводу, заканчивают тем, что просто выкидывают старый код и пишут его заново или решают никогда не обновлять его снова. Вместо того чтобы поддаться естественному желанию избежать болезненной процедуры, часто полезнее инвестировать в этот процесс, чтобы сделать его менее болезненным. Впрочем, выбор зависит от стоимости обновления, его ценности и ожидаемого срока службы проекта.

Суть устойчивости проекта заключается не только в преодолении первого крупного обновления, но и в достижении уверенности, что проект идет в ногу со временем. Устойчивость требует оценки влияния необходимых изменений и управления ими. Мы считаем, что достигли такой устойчивости во многих проектах в Google, в основном путем проб и ошибок.

Итак, чем отличается программирование короткоживущего кода от производства кода с более долгим ожидаемым сроком службы? С течением времени мы стали намного четче осознавать разницу между «работающим по счастливой случайности» и «удобным в сопровождении». Не существует идеального решения вышеназванных проблем. Это прискорбно, потому что длительное сопровождение ПО — это постоянная борьба.

Закон Хайрама

Если вы поддерживаете проект, который используется другими инженерами, то между «работающим по счастливой случайности» и «удобным в сопровождении» есть одно важное отличие, которое мы назвали *законом Хайрама*:

[1] Точка перехода во многом зависит от ваших приоритетов и предпочтений. Судя по нашему опыту, большинство проектов подходит к черте, когда обновление становится необходимостью, где-то через пять лет работы ПО. По более консервативным оценкам этот переход находится где-то между пятью и десятью годами службы.

> Если число пользователей API достаточно велико, неважно, что вы обещаете в контракте: любое наблюдаемое поведение системы будет зависеть от чьих-то действий.

По нашему опыту, эта аксиома является доминирующим фактором в любом обсуждении изменения ПО. На концептуальном уровне ее можно сравнить с энтропией: закон Хайрама обязательно должен учитываться при обсуждении изменений и сопровождения[1], так же как при обсуждении вопросов эффективности или термодинамики должна учитываться энтропия. Тот факт, что энтропия никогда не уменьшается, не означает, что мы не должны стремиться к эффективности. То, что закон Хайрама будет действовать применительно к сопровождению ПО, не означает, что мы не должны планировать или пытаться лучше понять это сопровождение. Можно смягчить последствия проблем, даже если мы знаем, что они никогда не исчезнут.

Закон Хайрама основан на практическом понимании, что даже при наличии самых лучших намерений, лучших инженеров и широкого круга методик проверки кода нельзя надеяться на полное соблюдение опубликованных контрактов или передовых практик. Как владелец API вы имеете *некоторую* свободу, четко понимая возможности интерфейса, но на практике сложность изменения также зависит от того, насколько полезным для пользователя является наблюдаемое поведение API. Если пользователи не зависят от него, изменить API будет легко. Но с течением времени и при достаточном количестве пользователей даже самые безобидные изменения *обязательно* что-то нарушат[2]. Анализируя ценность изменений, учитывайте трудности, связанные с поиском, выявлением и устранением нарушений, которые они вызовут.

Пример: упорядоченный хеш

Рассмотрим пример упорядочения итераций по хешу. Если вставить в хеш пять элементов, в каком порядке мы их получим?

```
>>> for i in {"apple", "banana", "carrot", "durian", "eggplant"}: print(i)
...
durian
carrot
apple
eggplant
banana
```

Большинство программистов знают, что хеш-таблицы не упорядочивают данные явно. Но почти никто не знает, что, *возможно*, хеш-таблица, которую они используют, возвращает содержимое в определенном порядке. Этот факт кажется непримечательным, но за последнюю пару десятилетий опыт использования хэша эволюционировал.

---

[1] Надо признать, что сам Хайрам собирался назвать этот закон «законом неявных зависимостей», но в Google предпочли более краткое название «закон Хайрама».

[2] См. комикс «Workflow» (http://xkcd.com/1172) на сайте xkcd.

- Атаки *переполнения хеша* (hash flooding)[1] стимулируют недетерминированный характер хранения данных в хеше.
- Потенциальный выигрыш от поиска усовершенствованных алгоритмов хеширования или хеш-контейнеров требует изменения порядка итераций в хеше.
- Согласно закону Хайрама, программисты по возможности пишут программы, зависящие от порядка обхода хеш-таблицы.

Если спросить эксперта: «Можно ли положиться на конкретный порядок обхода элементов в хеш-контейнере?» — он наверняка ответит: «Нет». Это правильный ответ, но слишком упрощенный. Более точный ответ мог бы звучать так: «Если код недолговечный и не предполагает будущих изменений в аппаратном или программном окружении или структуре данных, то это вполне допустимо. Но если известно, что код будет жить долго или нельзя гарантировать, что зависимости никогда не изменятся, то предположение неверно». Более того, даже если ваша собственная реализация не зависит от порядка хранения данных в хеш-контейнере, этот порядок может использоваться другим кодом, неявно создающим такую зависимость. Например, если ваша библиотека сериализует значения перед вызовом удаленной процедуры (RPC, remote procedure call), вызывающая сторона может оказаться в зависимости от порядка следования этих значений.

Это очень простой пример различия между «это работает» и «это правильно». Зависимость недолговечной программы от порядка хранения данных в контейнере не вызывает технических проблем. С другой стороны, для проекта, срок жизни которого преодолевает некоторый порог, такая зависимость представляет значительный риск: по прошествии времени кто-то или что-то может сделать этот порядок ценным. Ценность проявляется по-разному: как эффективность, безопасность или просто пригодность структуры данных для изменения в будущем. Когда ценность очевидна, взвесьте все за и против, выбирая между этой ценностью и проблемами, с которыми могут столкнуться разработчики или клиенты.

Некоторые языки намеренно изменяют порядок хеширования в каждой следующей версии библиотеки или при каждом запуске программы, чтобы предотвратить появление зависимости от этого порядка. Но даже в этом случае закон Хайрама преподносит сюрпризы, поскольку такое упорядочивание происходит с использованием генератора случайных чисел. Устранение случайности может нарушить работу кода пользователей, полагающихся на него. В любой термодинамической системе энтропия увеличивается, и точно так же к любому наблюдаемому поведению применим закон Хайрама.

Между кодами, написанными для «работы сейчас» и для «работы всегда», можно выделить четкие взаимосвязи. Рассматривая код как артефакт с переменным (в ши-

[1] Разновидность атак типа «отказ в обслуживании» (DoS, denial-of-service), при которых злоумышленник, зная внутреннюю организацию хеш-таблицы и особенности хеш-функции, может сформировать данные таким образом, чтобы снизить алгоритмическую производительность операций над таблицей.

роких пределах) временем жизни, можно определить стили программирования: код, зависящий от хрупких и недокументированных особенностей, скорее всего, будет классифицирован как «хакерский» или «хитрый», тогда как код, следующий передовым практикам и учитывающий возможность развития в будущем, вероятно, будет считаться «чистым» и «удобным для сопровождения». Оба стиля имеют свои цели, но выбор одного из них во многом зависит от ожидаемого срока службы кода. Мы привыкли использовать термин *программирование*, если характеристика «хитрый» — это комплимент, и *программная инженерия*, если слово «хитрый» имеет отрицательный оттенок.

Почему бы просто не «отказаться от изменений»?

Все наше обсуждение времени и реакции на изменения связано с неизбежностью изменений. Верно?

Как и все остальное, о чем говорится в книге, принятие решения об обновлении зависит от обстоятельств. Мы готовы подтвердить, что «в большинстве проектов, существующих достаточно долго, рано или поздно возникает необходимость обновления». Если у вас есть проект, написанный на чистом C и не имеющий внешних зависимостей (или имеющий только зависимости, гарантирующие стабильность в течение долгого времени, такие как стандарт POSIX), вы вполне можете избежать рефакторинга или сложного обновления. Разработчики языка C прикладывают значительные усилия, чтобы обеспечить его стабильность.

Но большинство проектов подвержены изменениям в базовой технологии. В основном языки программирования и среды выполнения меняются активнее, чем язык C. Даже библиотеки, реализованные на чистом C, могут меняться для поддержки новых функций и тем самым влиять на пользователей. Проблемы безопасности есть во всех технологиях, от процессоров до сетевых библиотек и прикладного кода. *Каждый* элемент технологии может стать причиной критической ошибки и уязвимости безопасности, о которых вы не узнаете заранее. Если вы не применили исправления для Heartbleed (http://heartbleed.com) или не смягчили проблемы с упреждающим выполнением, такие как Meltdown и Spectre (https://meltdownattack.com), потому что полагали (или были уверены), что ничего не изменится, последствия будут серьезными.

Повышение эффективности еще больше осложняет картину. Мы стараемся оснастить вычислительные центры экономичным оборудованием, разумно использующим процессоры. Но старые алгоритмы и структуры данных на новом оборудовании работают хуже: связанный список или двоичное дерево поиска продолжают работать, но растущий разрыв между скоростями работы процессора и памяти влияет на «эффективность» кода. Со временем ценность обновления аппаратного обеспечения может уменьшаться в отсутствие изменений в архитектуре ПО. Обратная совместимость гарантирует работоспособность старых систем, но не гарантирует эффективности старых оптимизаций. Нежелание или неспособность воспользоваться новыми аппаратными возможностями чреваты большими издержками. Подобные

проблемы сложны: оригинальный дизайн может быть логичным и разумным для своего времени, но после эволюции обратно совместимых изменений предпочтительным становится новый вариант (ошибок не было, но время сделало изменения ценными и востребованными).

Подобные проблемы объясняют, почему в долгосрочных проектах необходимо уделять внимание устойчивости независимо от того, влияют ли они непосредственно на нас или только на технологии. Изменения — это не всегда хорошо. Мы не должны меняться только ради перемен, но должны готовиться к изменениям и инвестировать в их удешевление. Как известно каждому системному администратору, одно дело — знать, что есть возможность восстановить данные с ленты, и совсем другое — знать, как это сделать и во что это обойдется. Практика и опыт — это двигатели эффективности и надежности.

Масштабирование и эффективность

Как отмечается в книге Бетси Бейер и др. «Site Reliability Engineering. Надежность и безотказность как в Google» (СПб.: Питер, 2019) (далее Site Reliability Engineering — SRE), производственная система в компании Google относится к числу самых сложных систем, созданных человеком. Формирование такой машины и поддержание ее бесперебойной работы потребовали бесчисленных часов размышлений, обсуждений и проектирования с участием экспертов со всего мира.

Большая часть *этой* книги посвящена сложностям, связанным с масштабированием в организации, которая производит такие машины, и процессам поддержания этой машины в рабочем состоянии в течение долгого времени. Давайте снова вернемся к понятию устойчивости кодовой базы: «База кода организации является *устойчивой*, если вы *можете* без опаски изменять то, что нужно, в течение срока ее службы». В данной формулировке под возможностями также понимаются издержки: если обновление чего-либо сопряжено с чрезмерными затратами, оно, скорее всего, будет отложено. Если со временем затраты растут сверхлинейно, это значит, что выполняемая операция не масштабируется[1] и наступит момент, когда изменения станут неизбежными. Когда проект вырастет вдвое и понадобится выполнить эту операцию снова, окажется ли она вдвое более трудоемкой? Найдутся ли человеческие ресурсы для ее выполнения в следующий раз?

Люди — не единственный ограниченный ресурс, который необходимо наращивать. Само ПО должно хорошо масштабироваться в отношении традиционных ресурсов, таких как вычислительная мощность, память, объем хранилища и пропускная способность. Разработка ПО также должна масштабироваться с точки зрения количества участвующих в ней людей и объема вычислительных ресурсов. Если стоимость вы-

[1] Всякий раз, когда в этой главе мы используем слово «масштабируемость» в неформальном контексте, мы подразумеваем «сублинейное масштабирование в отношении человеческих возможностей».

числений на тестовом кластере растет сверхлинейно и на каждого человека в квартал приходится все больше вычислительных ресурсов, это значит, что положение неустойчиво и скоро придется что-то изменить.

Наконец, самый ценный актив организации, производящей ПО, — кодовая база — тоже нуждается в масштабировании. При сверхлинейном росте системы сборки или системы управления версиями (VCS, version control system) (возможно, в результате увеличения истории изменений) может наступить момент, когда работать с ней станет невозможно. Многим аспектам, таким как «время для полной сборки», «время для получения новой копии репозитория» или «стоимость обновления до новой языковой версии», не уделяется должного внимания из-за того, что они меняются очень медленно. Но они с легкостью могут превратиться в метафорическую сварившуюся лягушку (https://oreil.ly/clqZN): медленно накапливающиеся проблемы слишком легко усугубляются и почти никогда не проявляются в виде конкретного момента кризиса. Только имея полное представление об организации в целом и стремясь к масштабированию, вы, возможно, сможете оставаться в курсе этих проблем.

Все, что организация использует для производства и поддержки кода, должно быть масштабируемым с точки зрения издержек и потребления ресурсов. В частности, все операции, которые организация должна выполнять снова и снова, должны быть масштабируемыми относительно человеческих усилий. С этой точки зрения многие распространенные политики не могут считаться хорошо масштабируемыми.

Плохо масштабируемые политики

Даже небольшой опыт разработки позволяет легко определять политики, плохо поддающиеся масштабированию, чаще всего по изменению объема работы, приходящемуся на одного инженера, при расширении компании. Если организация увеличится в 10 раз, увеличится ли в 10 раз нагрузка на одного инженера? Увеличится ли объем работы, которую он должен выполнять, при увеличении кодовой базы? Если на какой-то из этих вопросов будет дан положительный ответ, найдутся ли механизмы для автоматизации или оптимизации работы этого инженера? Если нет, значит, в организации есть явные проблемы с масштабированием.

Рассмотрим традиционный подход к устареванию (подробнее об устаревании в главе 15) в контексте масштабирования. Представьте, что принято решение использовать новый виджет вместо старого. Чтобы мотивировать разработчиков, руководители проекта говорят: «Мы удалим старый виджет 15 августа, не забудьте перейти к использованию нового виджета».

Такой подход хорошо работает в небольших проектах, но быстро терпит неудачу с увеличением глубины и широты графа зависимостей. Команды зависят от постоянно растущего числа виджетов, и любое нарушение в сборке может повлиять на рост компании. При этом вместо перекладывания хлопот, связанных с миграцией нового кода, на клиентов команды могут сами внедрить все необходимое, используя преимущества экономии от масштабируемого решения.