# # 1. Приложения, созданные для облака - концепция Cloud Native

> “Технологии Cloud Native позволяют создавать сложные системы в динамичной, современной среде частного, гибридного или коммерческого облака Cloud. Воплощают такой подход в жизнь контейнеры, сервисные сетки (service mesh), микросервисы, неизменная инфраструктура и декларативный способ управления ресурсами.“ Устав фонда Cloud Native Foundation, управляющего стандартами и общим направлением концепции.

Современная разработка классических приложений, вспомогательных сервисов, мобильных приложений и их серверных компонентов, и программного обеспечения в целом подразумевает постоянные изменения функциональности. Часто новые функции появляются несколько раз в день. При этом обновления, исправления ошибок и перезапуски не должны останавливать сервис и доступ к его функциям ни на секунду. Практически весь мир объединен глобальным скоростным доступом в Интернет, и одновременный доступ не только миллионов, но сотен миллионов пользователей к удачному приложению и сервису больше не является прерогативой технических гигантов, таких как Google, Apple и Facebook, и вполне доступен маленькому стартапу и индивидуальному программисту. Приложение должно быть готово к пиковой нагрузке, не уменьшая качество своего сервиса. Именно таким и будет приложение, с самого начала созданное для работы в облаке.

## ## Основные положения концепции Cloud Native

Возможность быстро и эффективно наращивать функциональность приложения, не переписывая и не ломая уже существующие функции и компоненты, а также их взаимодействия, требует особого подхода к разработке в общем, и к выпуску готовых релизов и их запуску на серверах в частности. Более того, для каждой задачи хорош свой инструмент, что в мире программирования означает, что для каждой задачи чуть лучше может подходить свой собственный язык, его экосистема, и набор библиотек.

Реализовать это возможно с помощью так называемых “микросервисов” (microservices), слабо связанных между собой компонентов единой системы или приложения. Они обмениваются данными через сеть, используя стандартные сетевые протоколы, как правило это протокол HTTP и стандарт REST. Добавление и обновление одного микросервиса никак не затрагивает остальные части системы. Микросервисы связываются друг с другом через сетевые порты и абстрактные протоколы, и каждый из них может быть написан на любом подходящем языке и технологии. Обновляются и перезапускаются они также независимо. Микросервисы часто противопоставляются единому, большому серверному приложению, так называемому “монолиту” (monolith).

Запуск на одной операционной системе разнородных приложений, написанных с помощью самых разнообразных технологий, как правило не сулит в себе ничего хорошего из-за конфликта системных зависимостей, библиотек и правил доступа. Эту проблему блестяще решают контейнеры (containers). Контейнеры - легкая форма виртуализации, они надежно изолируют приложения друг от друга, и в отличие от виртуальных машин, не требуют полной установки отдельной операционной системы. Запуск и остановка контейнеров практически мгновенна. Множество разнородных модулей и библиотек теперь смогут ужиться в одном сервере Linux, не мешая друг другу, и не требуя для запуска минут, как требуют полноценные виртуальные машины.

Одно из преимуществ приложения, разбитого на модули и микросервисы, работающих из собственных контейнеров - тонко настроенное горизонтальное масштабирование. Появляется возможность выделить наиболее нагруженную часть системы и запустить для ее сервисов и компонентов столько экземпляров, сколько необходимо для обработки текущей нагрузки. Для этого требуется практически неограниченная вычислительная мощность, растущая по требованию (ее еще называют эластичной) - эту мощь обеспечивают коммерческие провайдеры облака, такие как Google Cloud (GCP), Amazon Web Services (AWS), и Microsoft Azure.

Наконец, мощное, динамично меняющееся приложение, состоящее из сотен распределенных компонентов, соединенных между собой по сети, требует постоянного надзора и очень сложного управления, в том числе и для масштабирования и обновления компонентов. Здесь главную роль играет оркестратор контейнеров (orchestrator), самым популярным среди них без сомнения является Kubernetes. Для наблюдения трафика между компонентами, задержек, графиков исполнения запросов, и сбора и анализа журналов (logs) существуют целые комплексы программных решений, хорошо интегрированных с Kubernetes.

### ### Первые выводы

Подводя краткий итог, мы поставили задачу создать максимально гибкое, устойчивое к отказам, всегда доступное приложение, способное выдержать пиковые нагрузки, и увидели, как эта задача может быть решена. Именно описанный подход и методы являются основой приложения, созданного для работы и развертывания в облаке (cloud native). Вот его ключевые атрибуты:

* \* Микросервисы (microservices) как способ максимально возможной слабой связи между подсистемами приложения. По сути эти компонентная разработка, с прицелом на абсолютно независимый друг от друга процесс разработки, свободный выбор технологии, а также независимые выпуски новых версий и их развертывание на сервере.
* \* Контейнеры (containers) - легкая виртуализация в пределах одной операционной системы (как правило Linux), не требующая огромных тяжелых виртуальных машин, включающих в себя полную отдельную операционную систему. Контейнеры позволяют множеству микросервисов незаметно друг для друга работать на одном сервере.
* \* Эластичная, практически бесконечно доступная при необходимости вычислительная мощность, то есть новые и новые сервера для запуска контейнеров. Эти сервера должны обладать эффективным, автоматическим, легко воспроизводимым способом запуска и конфигурации. Как правило, это обеспечивают коммерческие провайдеры облаков, владеющие большими центрами данных. Большие организации могут себе позволить собственные центры данных с работающими на их основе частными облаками.
* \* Оркестровка и управление контейнеров, внутри которых находятся микросервисы, в одном или множестве экземпляров. Основным инструментов управления является сейчас Kubernetes, мощный, расширяемый оркестратор, способный управлять, обновлять, масштабировать, настраивать взаимодействие для сотен микросервисов. Оркестратор работает с набором физических или виртуальных серверов в кластере.
* \* Наблюдение за сложной сетью микросервисов и их взаимодействием, в том числе за состоящими из множества мелких сетевых вызовов транзакциями и комплексными операциями. Необходимы эффективные инструменты для сбора и анализа журналов (logs). В динамической, распределенной среде любой мелкий вызов может таить в себе причину общего сбоя.

Теперь давайте взглянем чуть подробнее, какие технологии, подходы и архитектура обеспечивают успех каждого из столпов концепции cloud native.

## ## Микросервисы - быстрый цикл разработки и постоянный выпуск

Микросервисы (microservices) - очередной виток развития компонентной разработки программных комплексов и приложений. Разбиение сложной задачи на составные более простые части, изоляция сложности, и поиск абстракций, позволяющих упростить и сделать задачу управляемой и решаемой - основа программирования в целом. Разбиение программы на пакеты, функции, классы, а затем и на совершенно независимо работающие друг от друга компоненты логически вытекает из анализа задачи.

Микросервисы - это компоненты вашего приложения, независимо друг от друга работающие в облаке и соединенные между собой не прямыми вызовами внутри одного процесса, а передачей данных по сети, используя заранее оговоренные протоколы (обычно HTTP или gRPC) и порты.

Эластичность и практически неограниченная вычислительная мощность облака дает нам возможность разбить приложение на логические компоненты и запускать их и управлять ими индивидуально. При необходимости легко увеличить пропускную способность приложения, увеличив количество экземпляров компонентов (работающих в виде микросервисов), испытывающих наибольшую нагрузку. Это так называемое горизонтальное масштабирование - при работе в облаке его возможности практически безграничны, при условии выбора удачной архитектуры приложения, дающей возможность разбить его вычислительные потоки на независимые части. Вертикальное масштабирование же подразумевает рост мощности одного сервера и его аппаратных возможностей, что крайне ограничено, и более того, самые мощные серверы обычно очень дороги.

Передача данных между микросервисами осуществляется по сети, по хорошо известным протоколам, поддерживаемыми практически всеми известными языками и их библиотеками. Микросервисы больше не являются частью единого проекта и репозитория в системе контроля версия, и разрабатывающие их команды теперь свободны делать любой выбор, эффективно позволяющий решить задачу, стоящую перед компонентом. Это открывает двери для быстро меняющегося мира технологий, и когда-то сделанный выбор архитектуры и языка для одного компонента больше не диктует того же новым компонентами и сервисам.

Гораздо меньший размер и менее связанная с другими компонентами функциональность позволяет программистам быстро проводить в жизнь новые идеи, рефакторинг кода, и пробовать новые подходы и процессы разработки. Разумный размер кода делает процесс разработки быстрым и удобным. Это же позволяет проще и настроить системы постоянного контроля качества и развертывания сделанных изменений на сервере (CI/CD, continuous integration and delivery), и сделать их работу быстрой, позволяя программистам быстро проверить, было ли их последнее изменение удачным.

Обратной стороной компонентной разработки в распределенной среде является отсутствие гарантии работоспособности - любой сетевой вызов, в отличие от вызова функции внутри единого процесса, подвержен отказам и сбоям, иногда в течение долгого времени. Размытые границы между микросервисами диктуют аккуратный выбор протоколов и передаваемых структур данных. Тестировать взаимодействие микросервисов, взаимодействующих по сети, иногда бывает крайне сложно.

Мы подробнее рассмотрим некоторые аспекты дизайна и разработки микросервисов и похожих на них компонентов в отдельной главе.

## ## Контейнеры - изоляция и гарантия неизменяемости версий

Мы только что увидели, как много потенциальных преимуществ может принести с собой разбиение приложения на независимые компоненты, или микросервисы. Особенно они важны для программистов, получающих намного больше свободы в своих экспериментах и выборе технологии. Однако запуск таких компонентов должен быть быстрым, а взаимодействующие технологии должны уживаться на одних и тех же серверах (в пределах кластера) без конфликтов и сложных конфигураций.

Решить эту задачу можно виртуальными машинами, запуская на них микросервисы. Однако виртуальная машина требует установки и запуска отдельной, самостоятельной операционной системы, и время ее запуска делает быстрое масштабирование и перезапуск компонентов практически невозможным. Как мы уже видели, эту задачу берут на себя контейнеры с их легкой виртуализацией с помощью возможностей Linux (другие операционные системы уже также их поддерживают). Время запуска контейнера практически то же, что для обычного процесса Linux, а изоляция приложений, их файловых систем, и ограничение их ресурсов мало чем отличается от полноценной виртуальной машины.

Все содержимое (файлы и зависимости приложения или его части), необходимое для запуска контейнеров, упаковывается в образы (image). Важным свойством образа является его неизменность (immutability), для каждой отдельной метки, или версии, этого образа. Поменять помеченный определенным образом образ с известной контрольной суммой уже невозможно. С практической точки зрения это означает, что созданная когда-то система, настроенная и работающая с определенным набором микросервисов, упакованных в образы для запуска в виде контейнеров, теперь всегда может быть заново воспроизведена в любой необходимый момент. Это важное качество воспроизводимости (reproducibility) гарантирует уверенность в текущем состоянии сложной, составной системы. Мы можем быть уверены в том, что работающая система не была запущена давно потерянным и никому больше не известным набором эзотерических скриптов.

В главе про контейнеры мы подробнее узнаем историю виртуальных машин и контейнеров, чуть подробнее взглянем на механизмы их работы, и на основной инструмент работы с контейнерами - Docker.

## ## Облако - неизменная эластичная инфраструктура. “Феникс” вместо “снежинки”

Эта книга для разработчиков, и для нас, после того как мы создаем серверное приложение или сервис, зачастую начинается довольно туманный период его реальной эксплуатации (production), на основных серверах компании. Классически управлением и запуском готового выпуска приложения заведуют администраторы, или операторы (operators), заведующие всеми деталями настройки и управления серверами. Операторы могут использовать совершенно отдельный от разработки процесс запуска, и свои собственные инструменты для управления настройками серверов.

Для разработчиков в подобном процессе эксплуатации исправление и анализ ошибок или нестандартных ситуаций может стать настоящей головной болью. Если управление эксплуатацией совершенно отделено от выпуска и тестирования новых версий, анализ и воспроизведение ошибок особенно сложны, так как настройки и версии операционных систем и их зависимостей могут значительно отличаться от тех, что используются при тестировании или локальной отладке.

Особенно тяжело управлять и анализировать поведение большой серверной системы в случае, если каждый сервер представляет собой уникальную “снежинку” (этот термин предложил Мартин Фаулер), то есть обладает уникальным набором настроек и конфигураций операционной системы и ее аппаратного обеспечения. В этом случае функциональность системы сливается с уникальностью сервера и становится очень трудно воспроизводимой, и довольно нестабильной.

Гораздо проще восстанавливать и копировать сервер, если он представляет собой “феникса”, способного быстро восстановиться из заранее подготовленного образа (“пепла”, если выражаться в терминах легенды). Еще лучше, если этот образ не бинарная копия диска, а список четких инструкций, по шагам восстанавливающих состояние сервера из известных проверенных компонентов. Эта инструкция хранится в системе контроля версий с историей всех изменений. Сервер, созданный по инструкции, всегда будет одинаковым, и таким образом, обеспечит неизменяемую инфраструктуру приложения (immutable infrastructure).

Если и разработчики, и операторы имеют доступ к легко читаемой, легко восстанавливаемой конфигурации своих серверных систем и кластеров, процесс передачи выпущенных сервисов из разработки в эксплуатацию становится прозрачным и легко поддерживаемым. Восстановление среды для тестирования или эксплуатации не представляет собой проблем. Слияние процессов разработки и управления иногда еще называют процессом DevOps (девопс, development + operations).

Облачные сервера как нельзя лучше подходят для реализации упомянутых выше “фениксов”, проверенных, неизменяемых серверов c прозрачной историей. Публичные провайдеры облака, такие как Amazon AWS, Google GCP и Microsoft Azure, создают свои виртуальные или реальные сервера из тщательно проверенных, безопасных версий известных операционных систем, которые не будут внезапно меняться в процессе работы сервера. Ваша команда DevOps затем может использовать подготовленные заранее инструкции для дополнительной автоматической настройки этих серверов.

В мире контейнеров все становится еще проще. Провайдеры облака как правило предоставляют оптимизированную операционную систему для запуска контейнеров (как правило особая версия Linux). Все зависимости и дополнительные настройки вы производите уже внутри самого контейнера, и указываете их при создании его образа, помечая его версией. Эта версия затем может многократно запускаться в виде контейнера и уже никогда не меняется. Управление сложными серверными конфигурациями значительно упрощается, и по сути, любой член команды, работающей в формате DevOps, способен без особого труда построить и восстановить любую конфигурацию системы, от тестирования до реальной эксплуатации, просто используя доступное в системе контроля версий описание развертывания и запуска системы.

Наконец, основный провайдеры облака как правило обладают мощными центрами данных. При недостатке вычислительной мощности и росте популярности вашего приложения вы сможете динамично расширить свой кластер, добавить в него новые сервера, и запустить на них необходимое количество экземпляров ваших микросервисов. Автоматизирует этот процесс оркестратор Kubernetes.

## ## Оркестровка Kubernetes - декларативное описание состояния

Получив неизменную, эластичную инфраструктуру, и все преимущества изоляции и быстрого запуска контейнеров, необходим инструмент, обладающей достаточной мощью гибкостью для управления ими. Оркестратор Kubernetes, созданный на основе проверенной годами в компании Google системы управления контейнерами Borg, обладает всем необходимым для запуска и управления сложными системами, развернутыми в облаке. Все основные коммерческие провайдеры облака, Google Cloud, AWS, Azure, российские #CloudMTS и остальные, в обязательном порядке предоставляют сервисы на основе Kubernetes.

Исходный код Kubernetes открыт и находится на GitHub, это один из самых популярных проектов с огромным количеством программистов и компаний, работающих над ним. Запустить Kubernetes возможно на своем собственном, частном облаке и кластере, а начать эксперименты можно и вовсе на своем ноутбуке, используя совершенно полные локальные реализации, такие как Docker for Desktop и Minikube.

Как мы увидели в уставе фонда Cloud Native, декларативное описание является неотъемлемой частью этой концепции. В случае с Kubernetes это означает, что вместо развертывания и запуска приложений (упакованных в образы контейнеров) явными командами из скриптов и терминала, предпочтительным является \*желаемое состояние кластера\* (desired state). Состояние описывает все системы, микросервисы и сетевые настройки системы, а управляющая система Kubernetes заботится о том, чтобы кластер представлял собой именно желаемое состояние. Состояние описано в файлах YAML. В главах, посвященных Kubernetes, мы узнаем все детали нем, и научимся описывать систему декларативным способом.

## ## Инструменты для сбора журналов и наблюдения

Развернув систему из множества микросервисов, общающихся между собой по сети, зачастую асинхронно, мы получим взрывной рост разнообразных условий, при которых система будет вести себя отлично от идеала. В отличие от монолитного приложения, журналы (logs) которого можно анализировать в относительном порядке, и не волноваться о сбоях вызовов внутри одного процесса, в распределенной, микросервисной среде каждый вызов может сорваться или внести каскадный сбой.

Вокруг концепции Cloud Native сложилась огромная, динамичная экосистема решений и продуктов, помогающих решить неизбежные проблемы распределенных асинхронных систем. Прежде всего это сервисные сетки (service mesh, смысл термина состоит в упорядочении и управлении сетевых переговоров микросервисов, но слово “сеть”, и мы используем сетку чтобы уменьшить путаницу), такие как Istio и Linkerd. Сервисные сетки решают многие проблемы, дают возможность наблюдать за сетевыми вызовами, строить графы, получить задержки, и многое другое. Системы сбора метрик, такие как Prometheus, позволяют интегрировать метрики вашей системы в единые центры наблюдения. Управление журналами, например Fluentd, справляется со сбором и упорядочением десятков разнообразных журналов, полученных от микросервисов. Мы еще раз вспомним эти инструменты, когда будем рассматривать микросервисы в следующей главе.

## ## Разработка на практике - 12 факторов облачного приложения

Теория и концепция Cloud Native, то есть приложений, созданных для облака, пока выглядит стройно и логично, и остальную часть книги мы посвятим практическому применению ее основных частей. Однако всегда интересно узнать “выжимку” опыта компаний, команд и программистов, которую уже попробовали разработку таких приложений, и увидели всю подноготную проблем, с которыми придется столкнуться - как мы знаем, в программировании многие неприятности скрываются именно в мелких деталях.

Команда облачного сервиса Heroku, популярного выбора для небольших команд и систем, собрала свои наблюдения в каталог из 12 факторов (12 factor app), наличие которых в дизайне и реализации системы резко повышает его шансы на успешную работу в облаке и простую поддержку готовой системы. Отсутствие этих факторов, или, что хуже, выбор противоположных решений, может впоследствии усложнить разработку и развертывание облачного приложения. Давайте посмотрим на эти факторы, и увидим, как они соотносятся с положениями концепции Cloud Native.

### ### 1 - Единая база кода

Весь код приложения, или микросервиса должен находиться в своем отдельном репозитории (GitHub или что-то еще). Использовать один репозиторий для нескольких сервисов не рекомендуется из-за возрастающей сложности сборки и связанности между компонентами системы. В главе про микросервисы мы подробнее узнаем, почему это хорошая мысль.

### ### 2- Явное описание и изоляция зависимостей

Облачное приложение ни в коем случае не должно рассчитывать, что на серверах кластера что-то будет доступно или предустановлено. Этот фактор отлично накладывается на рекомендуемый способ работы с контейнерами - вы всегда способны заново построить образ контейнера своего приложениях (обычно с помощью Dockerfile), и он включает в себя все необходимо для работы - любые библиотеки JAR, пакеты Node.js, и так далее.

### ### 3 - Управление конфигурацией

Гибкое приложение избегает включения любых элементов конфигурации в свой исходный код - это пароли, адреса баз данных, даже порты микросервисов-партнеров. Большую помощь в реализации этого фактора оказывает Kubernetes. Он, хоть и не может вас заставить вынести всю конфигурацию в переменные окружения (environment variables), предоставляет удобные инструменты, такие как “секреты” (secrets) и карты конфигурации (config maps). Они описываются декларативно, в файлах YAML. Меняя карты конфигурации, вы с легкостью можете развернуть свою систему в совершенно другом окружении, например для отладки или у нового клиента на его собственных серверах.

### ### 4 - Вспомогательные ресурсы через конфигурацию

Система, использующая дополнительные внешние системы и ресурсы, такие как базы данных, хранилища неструктурированных данных, почтовые и СМС-сервисы, в идеале максимально абстрагирует свои связи с ними. Выделение точного интерфейса для работы с ними, и вынесение в переменные окружения всех параметров для доступа и соединения с такими ресурсами поможет системе уменьшить количество зависимостей и легкость работы в разнообразных облаках и окружениях. Вновь, карты конфигурации Kubernetes отлично справятся. Для более сложных случаев можно описать ресурс в виде объекта Kubernetes (CRD, custom resource definition).

### ### 5 - Строгое разделение построения и запуска системы

Система не должна запускаться из непроверенных изменений в коде или конфигурации. Собранная система помечается версией или меткой (tag), все собранные бинарные и конфигурационные файлы доступны для перезапуска в случае проблемы. Этот фактор прекрасно обеспечивают образы (image) контейнеров - они неизменны после сборки, вы знаете историю версию в репозитории образов (обычно Docker Hub), и можете строго воспроизвести любое состояние системы, не откатывая никаких изменений в коде.

### ### 6 - Сервисы без состояния

Микросервисы облачного приложения в идеале не обладают вообще никаким состоянием и стараются не хранить никаких промежуточных результатов для выдачи другим серверам (stateless, share-nothing). Это позволяет добиться легкой масштабируемости и восстановления системы. Необходимо рассчитывать на динамичность облака и то, что любой сервер или диск может быть перезапущен в любую минуту. Данные должны храниться в специализированных сервисах для данных, обычно управляемых облаком - облачных базах данных (Cloud SQL, Amazon RDS), кэшах Memcached, и других. Как мы увидим, именно микросервисы без состояния намного проще создавать с помощью Docker и управлять Kubernetes.

### ### 7 - Доступ через сетевые порты

Микросервисы общаются через сетевые порты, обычно с помощью HTTP в стиле REST, отсылая данные в формате JSON или XML, или используют бинарный протокол GRPC. Если микросервис вызывает другие микросервисы-партнеры, адреса доступа к ним и их порты хранятся отдельно, в конфигурации. Данное требование идеально исполняется контейнерами, которые объявляют, по какому порту они будут ожидать соединений, и сервисами (service) Kubernetes, описывающим, как эти порты будут доступны в кластере. Все необходимое для работы HTTP сервера находится внутри контейнера (встроенные сервера, например Netty для Java).

### ### 8 - Масштабирование через запуск дополнительных экземпляров

Концепция микросервисов позволяет снизить количество ресурсов для четко выделенного компонента системы, и провести его точечное горизонтальное масштабирование - это возможно только в случае микросервисов без состояния (фактор 6). Kubernetes делает масштабирование, в том числе автоматическое в зависимости от загрузки системы, тривиальной задачей, и поддерживает желаемое количество экземпляров микросервиса.

### ### 9 - Быстрый запуск и надежная остановка

Микросервисы должны запускаться как можно быстрее, вновь в целях быстрой адаптации к возрастающей нагрузке, и более эффективного использования освобождающихся ресурсов. Легковесная виртуализация контейнеров Linux делает запуск новых контейнеров практически мгновенной, почти неотличимой от запуска обычного процесса. Отсутствие состояния и данных в основных компонентах бизнес-логики позволяет быстро их остановить, обновить, без потери данных и функциональности.

### ### 10 - Одинаковая среда разработки и эксплуатации

В больших распределенных системах, особенно если используется сложная авторизация, роли, базы данных, облачное хранилище данных, сразу же возникает вопрос как организовать среду разработки с похожим поведением, для отладки и проверки нового кода. Часто в целях экономии производственные облачные системы заменяют на менее мощные, или даже на локальные эмуляторы, которые отдаленно напоминают среду эксплуатации (production), но все же имеют множество мелких различий, и называют это средой разработки (dev environment).

Авторы 12 факторов яростно протестуют против подобного подхода - среда разработки, среда тестирования, и среда эксплуатации должны полностью совпадать, даже если придется платить за дополнительные ресурсы. В долгосрочном плане это сэкономит множество ресурсов на поиске проблем и сделает возможным более быстрый выпуск надежного нового кода. Я думаю многие из нас сталкивались не с полноценными средами разработки, и абсолютно не поддающимися воспроизведению в них ошибками реальной эксплуатации. Анализ ошибки в таком случае значительно усложняется.

Хотя контейнеры и Kubernetes не смогут автоматически предоставить вам идентичные среды, они сделают это намного проще, благодаря неизменности образов контейнеров, работающих в системе, и легкой переносимости конфигураций YAML. Следование факторам 2, 3, 4 и 6 также сделает создание идентичной среды разработки проще. Более того, если среды абсолютно идентичны, то любой член команды сможет выполнить развертывание, приближая команду к DevOps.

### ### 11 - Журналы logs в виде потока событий

В классических монолитных системах журналы пишутся на диск, в файлы, используется заранее выбранный формат, архивация и инструменты для их обработки (например, Log4J для Java). Ситуация кардинально меняется для контейнеров и системы из распределенных микросервисов. Контейнеры эфемерны и их файловая система пропадает вместе с их остановкой, разные технологии применяют различные форматы журналов, а понять что происходит с системой целиком по разнородным журналам крайне сложно.

В облачном приложении журналы не сохраняются и не обрабатываются. Все записи делаются в стандартный вывод (standard output), тот самый, что выводится в терминал при ручном запуске. Именно стандартный вывод используется в контейнерах и Kubernetes. Дополнительные решения (ELK, Fluentd), работающие под управлением Kubernetes, собирают журналы с различных микросервисов, анализируют и хранят их, и предоставляют инструменты для полного анализа.

### ### 12 - Администрирование как часть приложения

Дополнительные административные задачи, такие как миграция данных или удаление неудачных записей из распределенного кэша, могут исполняться только из среды эксплуатации, эти задачи тестируются вместе с построением и выпуском системы, и поставляются вместе. Уверенность в том, что дополнительное администрирование сделано проверенным способом и в нужной среде уменьшит количество ошибок. Неизменность контейнеров и легкий откат к предыдущим версиям развертываний (deployment) в Kubernetes позволят исправить неудачный выпуск системы.