КОНСПЕКТ ЛЕКЦИИ

CI/CD, стратегии ветвления Git и варианты деплоя

Длительность: 58 мин 29 с
Подготовлено: 2026-05-03 20:18

Кратко

> Лекция посвящена доставке кода в продакшн через CI/CD и связанным с этим практикам работы с Git. Лектор объясняет разницу между Continuous Integration, Continuous Delivery и Continuous Deployment, а также показывает, как эти подходы влияют на сборку, тестирование и выкладку артефактов. Отдельный большой блок посвящён стратегиям ветвления в Git: GitFlow, GitHub Flow, GitLab Flow, trunk-based и fork-based workflow. В конце разбираются стратегии деплоя — простой, Blue-Green, Canary, feature flags и rolling update — с их плюсами, минусами и сценариями применения.

Оглавление

Конспект

ГЛАВА 1

Введение в CI/CD и зачем он нужен

00:08 — 00:37

Лектор вводит тему доставки кода в продакшн и объясняет, что CI/CD нужен для автоматизации пути от разработки до выкладки. Поясняется, как раньше релиз был долгим, ручным и рискованным процессом с простоями и высоким шансом ошибки. Затем формулируется, что CI/CD повышает скорость, повторяемость и качество за счёт автоматизированных сборок и тестов.

Зачем нужен CI/CD

Лекция начинается с мотивации: раньше выкатка кода в production была сложной, долгой и часто требовала остановки сервиса или объявления технических работ. Это было связано с тем, что обновлять систему и одновременно пускать пользователей на нее физически не всегда было возможно. В результате релиз превращался в набор ручных операций, где участвовало много людей, а риск ошибки был высоким.

CI/CD нужен, чтобы автоматизировать путь от разработки до продакшена. Смысл в том, чтобы собирать артефакт, проверять его на разных стадиях и доставлять его в продакшн по заранее настроенному процессу. Лектор подчеркивает, что это один из главных процессов в современной доставке кода: важно не просто написать программу, а уметь надежно и быстро довести ее до работающего состояния в боевой среде.

Под CI/CD в лекции понимается специальный конвейер, который обслуживает наш код: он помогает собрать артефакт, проверить его и затем вкатить в нужную среду. До появления этих подходов было много ручной рутины: сборки делали вручную, копировали файлы вручную, ставили их на серверы вручную. Это замедляло релизы и увеличивало вероятность ошибок.

Лектор отдельно отмечает, что в старых подходах часто выкатывали сразу на production и сразу же там проверяли, что получилось. При этом на тестовых стендах до этого могли вообще не прогонять полный путь. Поэтому CI/CD рассматривается как способ повысить повторяемость, сократить ручные операции и уменьшить риск на боевом окружении.

ГЛАВА 2

CI, CD и различие между Delivery и Deployment

00:37 — 00:15

Разбирается расшифровка CI/CD и смысл каждой части: Continuous Integration отвечает за сборку и первичное тестирование, а CD — за доставку артефакта дальше по средам. Лектор отдельно подчёркивает разницу между Continuous Delivery и Continuous Deployment: в первом случае финальный деплой на продакшн ручной, во втором — автоматический. Также объясняется, почему в критичных системах автоматический деплой на прод используется осторожно.

Что такое CI и CD

CI/CD расшифровывается как Continuous Integration и Continuous Delivery или Continuous Deployment. Первая часть, CI, отвечает за автоматическую интеграцию изменений: сборку кода, первичное тестирование, проверку образа или артефакта, иногда даже контроль уязвимостей еще на этапе сборки. Вторая часть, CD, отвечает за доставку уже готового результата в другие среды.

Continuous Integration в лекции описывается как сборка и первичное тестирование. Это может быть не один тест, а несколько: например, сначала проверка во время сборки, потом дополнительные тесты уже после сборки. Лектор специально подчеркивает, что CI — это не один шаг, а набор автоматизированных процедур, которые подготавливают артефакт к дальнейшей доставке.

Continuous Delivery и Continuous Deployment — это разные вещи, хотя обе сокращаются как CD. Continuous Delivery означает, что весь путь автоматизирован почти полностью, но финальный деплой на production делается вручную. То есть человек в конце должен нажать кнопку «Задеплой». Это может быть релизный инженер или разработчик — не принципиально.

Continuous Deployment, наоборот, предполагает полностью автоматический выкат в production. Лектор отмечает, что для крупных и высококритичных систем это менее безопасно. Чем более бизнес-критична система, тем осторожнее обычно относятся к полной автоматизации выкладки в прод, потому что ошибка в таком сценарии может привести к серьезным последствиям.

ГЛАВА 3

Пайплайны, тесты и примеры CI/CD-схем

00:15 — 00:20

На примерах GitLab и типовых схем показывается, как выглядит пайплайн: коммит, сборка, юнит- и интеграционные тесты, стейджинг и выкладка в продакшн. Лектор отмечает, что pipeline — это цепочка автоматических действий, а тестирование может выполняться на разных этапах. В качестве иллюстрации приводятся и простые, и чрезмерно сложные реальные схемы CI/CD.

Как устроен пайплайн

Типичная схема CI/CD включает несколько этапов: код коммитится, потом начинается билд, затем прогоняются юнит-тесты, интеграционные тесты и другие проверки, после чего артефакт деплоится в промежуточные среды и уже потом в production. В лекции это показано как стандартный конвейер, где каждая стадия проверяет готовность следующей.

Для примера лектор приводит упрощенный GitLab-процесс: код коммитится, затем запускается сборка, на CI-этапе проходят unit-тесты и integration tests, после чего пайплайн переходит в CD и выкатывает артефакт в staging и другие среды, в том числе в production. Это иллюстрация того, что и CI, и CD — части одного общего процесса, но с разными задачами.

Важное свойство таких схем — наличие промежуточных тестовых стендов: acceptance-тесты, интеграционные тесты, smoke-тесты. Если система состоит из нескольких сервисов, нужно проверять не только каждый сервис отдельно, но и то, что их взаимодействие не ломается после выкладки. Поэтому пайплайн строится так, чтобы отловить ошибки раньше, чем они попадут к пользователю.

Лектор также показывает, что пайплайны могут быть как очень простыми, так и очень сложными. В простейшем виде это последовательность «протестировали — сбилдили — задеплоили». В более сложном — много этапов, несколько сред, ручные проверки, отдельные job'ы для мониторинга и контроля. Сама идея CI/CD здесь в том, что доставку и проверку кода можно формализовать как воспроизводимый процесс.

ГЛАВА 4

CI/CD в других областях и роль Git

00:20 — 00:37

Лекция расширяет идею DevOps на другие домены, прежде всего на MLOps, где модели и данные тоже рассматриваются как артефакты. Далее объясняется, почему CI/CD тесно связан с Git: код и описание пайплайнов обычно хранятся в репозитории, а значит, важны правила ветвления и работы с историей. Подчёркиваются практики аккуратной работы с коммитами, gitignore, code review и rebase.

CI/CD как основа для других областей

Лектор отмечает, что идея CI/CD оказалась настолько удобной, что ее стали переносить в другие области. Когда к базовому DevOps-подходу добавляют специфичный домен, получается отдельный класс практик: например, MLOps. В этом случае вместо обычного артефакта можно рассматривать модель машинного обучения.

Для MLOps логика похожая: модель можно циклически обучать, упаковывать как артефакт, хранить в виде образа или вольюма и затем автоматически деплоить. В этом смысле сам процесс остается тем же: что-то разработали или обучили, проверили, упаковали, выкатили, посмотрели на результат и снова вернулись к следующей итерации.

Лектор также замечает, что пайплайны — это не только практика, но и код. Часто описание того, как артефакт проходит свой путь, где запускаются тесты и как делаются выкладки, хранится в репозитории. Поэтому Git и стратегия ветвления становятся критически важными: они влияют не только на хранение исходников, но и на весь процесс доставки.

Отсюда возникает связь между CI/CD и GitOps-подходами. Если инфраструктура и пайплайны тоже описаны как код, то структура репозитория, коммиты, ветки и правила мержа начинают напрямую определять, насколько удобно и безопасно работать с доставкой изменений.

ГЛАВА 5

Стратегии ветвления Git: обзор и сравнение

00:25 — 00:37

Лектор вводит понятие стратегии ветвления и объясняет, что она определяет способ организации разработки и доставки кода. Сравниваются несколько моделей: OneBranchFlow, GitFlow, GitHub Flow, GitLab Flow, trunk-based и fork-based workflow. Для каждой подчеркивается компромисс между простотой, управляемостью, историей релизов и удобством интеграции с процессами Ops.

Почему стратегия ветвления важна

После обсуждения CI/CD лектор переходит к Git, потому что процесс доставки кода обычно неразрывно связан с тем, как организованы ветки и коммиты. Стратегия ветвления — это набор правил о том, как команда разрабатывает ПО, как пушит изменения, как интегрируется в общую ветку и как готовится релиз.

Одна из причин важности Git — в нем хранится код, который потом будет собираться CI. Обычно CI реализуется на стороне хранилища кода, например, в GitLab CI. Поэтому Git-структура влияет на то, как именно запускаются пайплайны и какие сущности репозитория нужно учитывать.

Лектор выделяет несколько типов стратегий. Самая простая — OneBranchFlow, когда все идет в одну ветку. Это подходит для маленьких личных проектов или ботов, где нет строгих требований к релизам и командной разработке. В такой схеме можно и разрабатывать, и деплоить из одной ветки без усложнений.

Далее идет GitFlow — одна из самых популярных, но не всегда самых эффективных схем. Ее идея: есть стабильная ветка Main, которая хранит самый актуальный, stable-код, но активно не используется. Основная рабочая ветка — Develop. От Develop отводятся Feature-ветки под отдельные задачи, которые затем вмерживаются обратно.

ГЛАВА 6

Практические замечания по Git и выбор стратегии

00:31 — 00:38

На вопрос из аудитории лектор поясняет разницу между ветками, тегами и релизными ветками в GitFlow и GitLab Flow. Отдельно обсуждается, зачем сохранять релизные ветки и когда это полезно для истории, сравнения версий и поиска потерянных изменений. В ответе также даётся практический совет: для новых рабочих проектов проще всего стартовать с GitHub Flow.

Практика работы с Git

Лектор отдельно говорит о хороших и плохих практиках в Git. Среди плохих — пушить все в один main без веток в командной разработке, работать одному над всем проектом без распределения ответственности и использовать бессмысленные коммит-месседжи вроде test, debug, final try. Такие подходы ухудшают читаемость истории и делают поддержку проекта сложнее.

Среди хороших практик — частые и мелкие коммиты вместо одного большого, который копился весь день. Большой коммит плохо читается, плохо откатывается и неудобен для CI/CD, где важно уметь быстро вернуться к предыдущему состоянию. Лектор также советует использовать осмысленные названия коммитов, gitignore и следить за тем, чтобы в репозиторий не попадали лишние файлы.

Отдельный акцент сделан на секретах: если логины и пароли однажды попали в Git, они остаются в истории навсегда. Это один из самых неприятных сценариев, поэтому игнорирование секретов и правильная настройка репозитория критически важны. Кроме того, полезны code review и rebase, если ваша ветка уже отстала от основной и нужно привести ее к актуальному состоянию перед merge.

Дальше лектор сравнивает несколько стратегий ветвления. GitFlow сложен и медленен, зато хорошо подходит для релизного процесса с отдельными release-ветками и hotfix-ветками. GitHub Flow проще: есть main и feature-ветки, изменения вмерживаются в main, который считается стабильным и актуальным. GitLab Flow — промежуточный вариант, где есть мастер/маин, отдельные ветки для подготовки релиза и ветки, которые остаются в истории для отслеживания конкретных релизов. Trunk-based — самая строгая и быстрая схема, где все разрабатывается почти в одной основной ветке, а короткоживущие ветки используются минимально.

ГЛАВА 7

Варианты деплоя: от простого до Blue-Green и Canary

00:39 — 00:47

После разговора о ветвлении лекция переходит к стратегиям деплоя как отдельному слою поверх CI/CD. Сначала разбирается простой деплой и частичное обновление как базовые, но рискованные подходы. Затем подробно объясняются Blue-Green Deploy и Canary Deploy: их механика, преимущества, стоимость и способы переключения трафика.

Простой деплой и его ограничения

Переходя к деплою, лектор подчеркивает, что CI/CD — это только конвейер доставки артефактов, а сами способы выкладки тоже бывают разными. Самый простой вариант — обычный deploy: обновили сервисы до новой версии, и если что-то сломалось, узнаем об этом только после выкладки. Такой способ подходит для некритичных систем, где допустим простой и быстрое восстановление.

Похожий вариант — частичное обновление, когда не весь набор сервисов выкатывается сразу, а по очереди. Это помогает уменьшить риск, связанный с зависимостями между сервисами, но все равно не дает полноценной защиты от ошибок. Откат при этом остается не очень быстрым.

Blue-Green Deploy строится на двух идентичных средах, или двух «плечах». Одно плечо в каждый момент времени простаивает, а другое обслуживает трафик. Сначала обновляют простаивающее плечо, проверяют его, потом резко переключают на него трафик и обновляют второе. Это дает возможность мгновенного отката и полноценного тестирования на production без участия пользователей.

Минус Blue-Green — высокая стоимость: нужно держать две полные копии прода, причем одна все время простаивает. Кроме того, переключение не абсолютно мгновенное: если в момент переключения выполняется тяжелый запрос, он может пострадать. Поэтому такой вариант часто используют в критичных инфраструктурах, где важны надежность и быстрый rollback.

ГЛАВА 8

Feature flags и rolling update

00:49 — 00:57

Лектор показывает, что feature flags позволяют выкатывать код с включением и выключением функционала внутри приложения, не полагаясь на несколько копий среды. Затем объясняется rolling update как постепенное обновление реплик с проверкой здоровья через пробы, при котором трафик переводится только после готовности всех экземпляров. Обсуждаются ограничения: необходимость stateless-приложений и риск при высокой нагрузке или неудачном совпадении с апдейтом.

Canary и feature flags

Canary-деплой похож на Blue-Green, но более гибкий и дешевый. Не обязательно иметь целое второе плечо: достаточно поднять новую версию сервиса рядом со старой и постепенно перенаправлять часть трафика на новую версию. Например, сначала 25% пользователей идут на новую версию, а 75% остаются на старой. Если все нормально, долю постепенно увеличивают.

Главное преимущество Canary — низкий риск: система тестируется на реальных пользователях, но только на малой доле трафика. Поэтому при проблеме откатить изменения можно быстро. Минус — сложность реализации: нужна умная настройка балансировки, мониторинга и метрик, чтобы отслеживать поведение обеих версий и управлять распределением трафика.

Лектор отдельно поясняет, что в Docker Compose можно имитировать такое поведение через реплики: например, один и тот же сервис поднять в двух экземплярах и распределить нагрузку в соотношении 2 к 10 и 8 к 10. Тогда балансировка будет работать по всем репликам, но большая часть запросов пойдет на старую версию. Это похоже на искусственно созданный Canary.

Feature flags — более точечный механизм. Внутри одного и того же кода включают или выключают функции через флаги, например, `Enable` или `false` для интеграции с RabbitMQ. Это удобно, когда физически невозможно держать две копии сервиса. Rolling update — еще один распространенный способ: реплики обновляются по очереди, старая и новая версии сосуществуют, а трафик на новую версию пускается только после успешных проверок. Лектор отмечает, что для rolling update приложения должны быть stateless, а система деплоя должна уметь проверять готовность каждой реплики, например, через health checks или пробы в Kubernetes.

Ключевые цитаты

«И уже потом это попадает в продакшн. И вот как раз за это отвечает процесс CICD.»

00:10 Это базовое определение CI/CD как процесса доставки артефакта в production.

«Непрерывная интеграция – это у нас сборка и первичное тестирование.»

00:13 Это короткая и запоминающаяся формулировка смысла CI.

«Continuous Delivery – это у нас, когда мы берем результат нашего CI и везем его куда-нибудь.»

00:13 Фраза фиксирует отличие CD как этапа доставки уже готового результата.

«Continuous Delivery предполагает только ручной деплой на продакшен.»

00:13 Здесь дана ключевая граница между Delivery и Deployment.

«Continuous Deployment – это как раз про то, что у нас и выкатка в прод тоже автоматизирована.»

00:14 Это второе важнейшее определение, без которого нельзя понять различие CD-терминов.

«Если у вас какой-то мелкий проект, какой-то pet project вы дома делаете, конечно, ничего страшного в этом нет. Но мы говорим про нормальную среду типа команды и так далее.»

00:21 Эта мысль задает контекст: многие практики Git оцениваются по-разному для личного и командного проекта.

«Мы стараемся коммитить более часто и более мелкими.»

00:22 Это одно из главных практических правил работы с Git, влияющее на откат и читабельность истории.

«Git ignore ... можно записать, чтобы у нас не попало туда что-то лишнее.»

00:23 Фраза напоминает о важной гигиене репозитория и защите от утечки секретов.

«Не забывать делать ребейс, условно говоря, потому что если вы отвели ветку от какой-то ветки, и пока вы там в ней работали, основная ветка уже далеко убежала, то всегда хорошим тоном считается ребейзинг.»

00:24 Это важное правило для поддержания актуальности веток перед merge.

«У нас есть, соответственно, ветка Main, которая содержит самый актуальный код из возможных. Но, тем не менее, этот Main мы никак не используем вообще.»

00:26 Это центральная идея Git Flow: main хранит стабильное состояние, но не является рабочей веткой.

«Когда мы видим, что какое-то супер мелкое изменение... в таких случаях как раз существует сущность hotfix.»

00:28 Это объясняет, когда нужен hotfix и почему он выделяется отдельно от обычного релизного процесса.

«Ветки короткоживущие, с какими-то супер мелкими этими. Соответственно, здесь мы стараемся избежать большого количества веток, но добавляем множественные ключи, чтобы мы могли включать, выключать нужные фичи.»

00:34 Это суть trunk-based разработки и ее связь с feature flags.

«Самый простой деплой, который существует, это, в общем-то, просто деплой.»

00:40 Это вводная формулировка к сравнению разных стратегий деплоя.

«Blue-Green Deploy. То есть, у нас есть две идентичные среды.»

00:42 Это базовое определение Blue-Green deployment.

«Можно мгновенно откатить, если вы вдруг переключили, что-то пошло не так, быстренько откатили обратно на плечо, которое еще не обновилось.»

00:43 Это ключевое преимущество Blue-Green: быстрый rollback.

«Идея в чем? Что мы... можем просто поднять рядом вторую версию сервиса и переключать трафик.»

00:45 Это коротко объясняет принцип Canary deployment.

«Это удобнее, это дешевле, чем Bluegreen. Это быстро откатить обратно.»

00:45 Здесь сформулированы главные плюсы Canary по сравнению с Blue-Green.

«С одной стороны, не совсем правильно считать полноценными стратегиями деплоя, потому что здесь это больше на уровне бизнеса работает, на уровне фич, а не на уровне техники.»

00:49 Это важное уточнение о feature toggles и feature flags как о механизме не только деплоя, но и экспериментов.

«Сервисы состоят из нескольких реплик, мы их обновляем постепенно.»

00:52 Это ядро Rolling Update как стратегии постепенного обновления.

Глоссарий

CI/CD

Подход к автоматизации сборки, тестирования и доставки кода в продакшн; включает Continuous Integration и Continuous Delivery/Deployment.

Continuous Integration

Часть CI/CD, где код автоматически собирается и проходит первичное тестирование.

Continuous Delivery

Режим, в котором артефакт доводится до готовности к продакшн-деплою, но финальный выкладочный шаг остаётся ручным.

Continuous Deployment

Режим, где выкладка в продакшн тоже автоматизирована и происходит без ручного подтверждения.

Артефакт

Результат сборки, который дальше тестируется и доставляется в другие среды, вплоть до продакшна.

Пайплайн

Последовательность автоматических шагов, по которым артефакт проходит сборку, тестирование и деплой.

Стейджинг

Промежуточная среда, где проверяют собранный артефакт перед продакшном.

Продакшн

Боевая среда, где работает конечный пользовательский сервис.

GitFlow

Стратегия ветвления с отдельными develop, feature, release и hotfix ветками и релизами через main.

GitHub Flow

Простая стратегия, где есть основная ветка и короткоживущие feature-ветки, которые вмерживаются обратно.

GitLab Flow

Вариант ветвления, где релизная ветка создаётся от стабильной версии, проходит проверку и потом сливается обратно.

Trunk-based development

Стратегия, при которой разработка ведётся в одной основной ветке с короткими ветками и частыми вливаниями.

Fork-based workflow

Модель, где изменения в open source обычно делаются через форк репозитория и pull request между репозиториями.

Blue-Green Deploy

Стратегия деплоя с двумя идентичными средами, между которыми переключают трафик после проверки обновлённой копии.

Canary Deploy

Стратегия, где новую версию выкатывают на небольшую долю трафика и постепенно увеличивают охват.

Feature flags

Флаги внутри приложения, которые позволяют включать и выключать функциональность без отдельного деплоя разных версий.

Rolling update

Постепенное обновление реплик сервиса, при котором новая версия вводится поэтапно, а трафик переводится после успешных проверок.

Пробы

Проверки готовности и здоровья, которые оркестратор использует, чтобы решить, можно ли пускать трафик на реплику.

Hotfix

Срочная короткая ветка для микроскопического исправления, которую создают прямо от main и быстро возвращают обратно.

Вопросы для самопроверки

1. **Что такое CI/CD и какую проблему он решает?**

   ОТВЕТ. CI/CD автоматизирует сборку, тестирование и доставку кода, сокращая ручной труд, риск ошибок и простои при релизе.

2. **В чём разница между Continuous Delivery и Continuous Deployment?**

   ОТВЕТ. Delivery заканчивается ручным нажатием кнопки на этапе выкладки в прод, а Deployment доводит до автомата и сам финальный деплой.

3. **Почему CI/CD особенно полезен при наличии нескольких стендов?**

   ОТВЕТ. Потому что одинаковый пайплайн позволяет повторяемо проверять артефакт на разных средах и заранее ловить ошибки.

4. **Зачем нужна стратегия ветвления в Git?**

   ОТВЕТ. Она задаёт правила организации разработки, вливания изменений и подготовки релизов, что упрощает интеграцию с CI/CD.

5. **Чем GitFlow отличается от GitHub Flow?**

   ОТВЕТ. GitFlow использует develop, feature, release и hotfix ветки и больше ориентирован на релизы, а GitHub Flow проще: основная ветка и короткие feature-ветки.

6. **Зачем в GitFlow нужны release-ветки?**

   ОТВЕТ. Чтобы подготовить релиз, внести финальные исправления, сравнить версии и затем слить результат обратно в develop и main.

7. **Когда уместен trunk-based development?**

   ОТВЕТ. Когда команда умеет быстро и аккуратно интегрировать изменения в одну основную ветку и хочет минимизировать ветвление.

8. **В чём идея Blue-Green Deploy?**

   ОТВЕТ. Есть две идентичные среды: одна активна, другая обновляется и проверяется, после чего трафик переключается на новую.

9. **Как работает Canary Deploy?**

   ОТВЕТ. Новая версия получает лишь часть трафика, затем доля увеличивается, если метрики и поведение в норме.

10. **Что дают feature flags по сравнению с отдельными средами?**

    ОТВЕТ. Они позволяют включать и выключать функционал внутри одной версии кода без необходимости держать несколько копий среды.

11. **Как работает rolling update?**

    ОТВЕТ. Реплики обновляются по одной, а трафик переводится на новую версию только после прохождения проверок готовности.

12. **Почему rolling update требует stateless-приложений?**

    ОТВЕТ. Потому что старые и новые реплики должны сосуществовать одновременно, не ломая состояние и совместную работу сервиса.

Что изучить дополнительно

Infrastructure as Code и описание пайплайнов кодом

Лектор упоминал, что пайплайны и инфраструктурные настройки описываются кодом, но не разбирал это отдельно. Это важно, чтобы понимать, как CI/CD и GitOps реализуются на практике.

GitOps

Понятие было связано с ожиданиями Ops и релизными ветками, но не раскрыто подробно. Стоит изучить, как Git становится источником истины для инфраструктуры и деплоя.

Kubernetes probes

Пробы были упомянуты как механизм проверки готовности контейнеров при rolling update, но без деталей. Нужно понять readiness/liveness-проверки и их влияние на оркестрацию.

Автоскейлинг в сочетании с деплоем

Лектор кратко отметил конфликт rolling update и автоскейлинга. Полезно изучить, как HPA/VPA взаимодействуют с обновлением реплик и балансировкой нагрузки.

Балансировка трафика и reverse proxy

В контексте blue-green и canary упоминались балансировщики и HAProxy, но без детальной схемы. Это нужно, чтобы понимать, где именно реализуется переключение и дробление трафика.

Docker Compose v2 replicas

Пример с репликами был приведён как упрощённая реализация canary-подхода. Стоит изучить, как Compose распределяет трафик и как это связано с внешним балансировщиком.

Совместимость версий и миграции БД при деплое

В лекции отмечено, что старое и новое приложение должны сосуществовать, особенно на уровне базы данных. Нужно отдельно разобрать схемы миграций и backward compatibility.