КОНСПЕКТ ЛЕКЦИИ · LECTA

CI/CD, стратегии ветвления и стратегии деплоя

Длительность: 58 мин 29 с
Подготовлено: 2026-06-07 14:36

Кратко

> Лекция посвящена тому, как доставлять код и модели в продакшн с помощью CI/CD, а также каким образом выбирать стратегию ветвления и деплоя. Лектор объясняет разницу между Continuous Integration, Continuous Delivery и Continuous Deployment, показывает, зачем нужны автоматизация, тестирование на разных стендах и воспроизводимость процесса. Отдельно разбираются основные Git-подходы: One Branch Flow, Git Flow, GitHub Flow, GitLab Flow, trunk-based development и fork-in-flow. Во второй части лекции рассматриваются стратегии деплоя: обычный, частичный, Blue-Green, Canary, A/B-тестирование, feature flags и rolling update.

Оглавление

1. Введение в доставку кода и смысл CI/CD 02:04
2. Continuous Integration и Continuous Delivery/Deployment 11:56
3. Как выглядит CI/CD-пайплайн на практике 15:44
4. Git, коммиты, ревью и ветвление 20:14
5. Стратегии ветвления: One Branch Flow, Git Flow, GitHub Flow, GitLab Flow 25:49
6. Trunk-based development и fork-in-flow 34:44
7. Стратегии деплоя: от простого обновления к Blue-Green и Canary 39:30
8. A/B-тестирование, feature flags и rolling update 49:18
9. Ответы на вопросы и практические рекомендации 55:30

Конспект

ГЛАВА 1

Введение в доставку кода и смысл CI/CD

02:04 — 11:56

Лектор вводит тему доставки артефактов в продакшн и объясняет, почему CI/CD стал ключевым процессом в DevOps. Разбирается, как раньше релизы были медленными, ручными и рискованными, а затем процесс стал автоматизироваться. Вводится базовая схема: разработка, тестирование, сборка и доставка в продакшн.

Смысл доставки кода

Лектор вводит тему как процесс доставки нашего кода из среды разработки в продакшн. До этого уже обсуждались DevOps-концепции, контейнеризация и понятие артефакта. Ключевой вопрос здесь — как собрать этот артефакт и безопасно довести его до боевого окружения.

Зачем нужен CI/CD

CI/CD — это специальная штука, которая закрывает путь от разработки до продакшна. В типичном процессе есть среда разработки, затем сборка артефакта, затем его тестирование на разных этапах и стендах, и только после этого — попадание в production. Именно этот конвейер и автоматизирует CI/CD.

Как было раньше

На ранних этапах разработки, когда подобных практик еще не было, любой выкатываемый на прод релиз часто сопровождался простоем. Иногда приходилось заранее ставить заглушку или объявлять технические работы, потому что одновременно обновлять систему и пускать пользователей было технически сложно или невозможно.

Ручной процесс и его проблемы

Процесс релиза был долгим и рутинным: ручные сборки артефактов, ручная доставка, копирование, установка, участие множества людей. Отсутствие автоматизации приводило не только к низкой скорости, но и к высокой вероятности ошибок. Особенно опасно это было из-за слабого тестового покрытия и привычки проверять все уже почти на проде.

Что меняет CI/CD

Появление CI/CD сделало процесс заметно проще: сборка, тестирование и доставка стали повторяемыми и автоматизированными. Благодаря этому уменьшается число ошибок до попадания на Prod, повышается качество кода и растет скорость релизов. Вместо постоянной ручной рутины появляется воспроизводимый процесс, где артефакт проходит через заранее заданные стадии и стенды.

ГЛАВА 2

Continuous Integration и Continuous Delivery/ Deployment

11:56 — 15:44

Объясняется, что CI отвечает за сборку и первичное тестирование, а CD — за дальнейшую доставку готового артефакта. Лектор отдельно разводит Continuous Delivery и Continuous Deployment: в первом случае финальный деплой на прод выполняется вручную, во втором — автоматически. Подчеркивается, что автоматизация повышает скорость и качество, но степень автоматизации зависит от критичности системы.

Continuous Integration

Continuous Integration — это непрерывная интеграция, то есть автоматическая сборка и первичное тестирование. Лектор подчеркивает, что CI отвечает за действия вроде автоматической сборки, тестирования образа и первичной отправки артефакта в следующий этап. Иногда в CI входит сразу несколько проверок: например, тестирование уязвимостей во время сборки и отдельная проверка уже готового образа.

Continuous Delivery и Continuous Deployment

CD в CI/CD может означать либо Continuous Delivery, либо Continuous Deployment, и это разные вещи. Continuous Delivery — это когда весь путь автоматизирован почти до конца, но последний шаг на production остается ручным: человек нажимает кнопку деплоя, когда все готово. Continuous Deployment — это уже полностью автоматическая доставка, включая выкладку в прод.

Почему delivery и deployment не одно и то же

В Continuous Delivery мы автоматизируем все, что можно, до самого последнего этапа. Идея в том, что релиз должен быть готов, проверен и только потом вручную разрешен к попаданию в production. В Continuous Deployment этот барьер снимается, поэтому релиз уходит в прод без ручного подтверждения.

Ограничения полной автоматизации

Чем система крупнее и критичнее, тем менее реалистичен Continuous Deployment. Для business-critical систем полная автоматизация выкладки опасна: при ошибке ущерб будет слишком большим. Поэтому полностью автоматическая доставка уместнее в небольших, менее критичных проектах, где сбой быстро чинится.

Что дает CI/CD на практике

Главная польза CI/CD в том, что большая часть работы автоматизирована, а артефакт проходит через разные стенды — production и его копии. Это позволяет ловить ошибки раньше, чем они попадут к пользователям, и менять количество, тип и конфигурацию стендов гибко под задачу. Автоматизация ускоряет процесс и делает его более предсказуемым.

ГЛАВА 3

Как выглядит CI/CD-пайплайн на практике

15:44 — 18:11

На примерах показывается типовой пайплайн: код коммитится, билдится, тестируется, затем выкатывается на dev, staging и prod. Лектор отмечает, что в реальных проектах пайплайны могут быть очень сложными и включать много стендов и параметров. Отдельно упоминается, что CI/CD — это не только автоматизация, но и возможность удобно откатывать и сравнивать релизы.

Типичный CI/CD-пайплайн

На практике CI/CD выглядит как последовательность этапов: код коммитится, затем начинается build, дальше прогоняются unit-тесты, интеграционные тесты и дополнительные проверки. После завершения CI начинается CD: артефакт деплоится на stage и затем в другие среды, включая production.

Пример из GitLab

Лектор приводит схему из GitLab: сначала что-то протестировали, потом собрали, затем задеплоили в dev, потом снова собрали и задеплоили в prod. Отдельно могут быть job’ы для мониторинга и настройки. Это очень упрощенный, но показательный вариант того, как CI/CD реализуется в реальных системах.

Стенды и повторяемость

Смысл пайплайна в том, что у нас есть production и его условные копии — тестовые стенды. На них можно проверить поведение артефакта и примерно предсказать, как он поведет себя на боевом окружении. За счет повторяемости на множестве стендов уменьшается риск неожиданностей.

Почему это удобно

Автоматизированный пайплайн удобен тем, что снимает зависимость от локальной среды и ручных действий. Новую версию можно быстро собрать и пустить по уже отработанным маршрутам. Лектор отдельно отмечает, что такой подход позволяет гибко добавлять и убирать стенды, а также менять их настройки под задачу.

Реальные сложности

В реальности CI/CD может быть сильно усложнен множеством нюансов: параметризацией, большим числом сред, дополнительными проверками и ручными исключениями. Лектор шутливо показывает и «идеальный» путь, и хаотичный процесс с ручными правками на стенде, которые забыли закоммитить. Этот пример подчеркивает, зачем нужны автоматизация и дисциплина в пайплайне.

ГЛАВА 4

Git, коммиты, ревью и ветвление

20:14 — 23:44

Лекция переходит к тому, почему CI/CD почти всегда связан с Git и почему важна стратегия ветвления. Лектор советует чаще коммитить мелкими изменениями, использовать осмысленные commit message, gitignore, code review и rebase. Подчеркивается, что история репозитория влияет на возможность безопасной доставки и отката изменений.

Почему CI/CD связан с Git

Лектор отдельно подчеркивает, что CI/CD почти всегда тесно связан с Git. Во-первых, код где-то хранится, и обычно именно в Git-репозитории запускается CI. Во-вторых, сами пайплайны и правила доставки тоже описываются кодом, то есть инфраструктура и процесс доставки становятся частью репозитория.

Стратегия ветвления как часть процесса доставки

Если процесс описан кодом, то стратегия ветвления становится важнейшей частью организации работы. Для CI/CD важно не только что мы коммитим, но и как мы движемся между ветками, как готовим релизы и как откатываемся. Поэтому правила ветвления — это не только про разработку, но и про доставку кода.

Коммиты и их качество

Лектор приводит негативный пример: пушить все в один main — плохая практика для нормальной командной среды, хотя для pet project это допустимо. Также он рассказывает про слишком большие коммиты: один длинный коммит на русском с огромным текстом в message плохо читается и плохо откатывается. Лучше делать частые и мелкие коммиты.

Хорошие практики в Git

Рекомендуется использовать naming convention для коммитов, Gitignore для исключения лишних файлов и секретов, а также code review. Важна и привычка делать rebase, если рабочая ветка отстала от основной: перед merge нужно привести ее к актуальному состоянию. Это уменьшает конфликты и делает историю чище.

Зачем это нужно в CI/CD

Все эти практики важны потому, что в CI/CD особенно критично уметь быстро и надежно откатывать состояние обратно. Чем более аккуратна история коммитов и ветвлений, тем проще поддерживать стабильность релизов, сравнивать изменения и находить источник проблем.

ГЛАВА 5

Стратегии ветвления: One Branch Flow, Git Flow, GitHub Flow, GitLab Flow

25:49 — 31:44

Разбираются популярные модели ветвления: One Branch Flow для простых pet-проектов, Git Flow для формально организованных релизов, GitHub Flow как более простой и практичный вариант, а также GitLab Flow как промежуточная схема. Поясняется, чем отличаются main, develop, feature-, release- и hotfix-ветки, и когда они нужны. Лектор отдельно отмечает, что выбор стратегии зависит от команды, частоты релизов и зрелости процессов.

One Branch Flow

Самая простая стратегия — One Branch Flow. Лектор описывает ее как случай pet project или небольшого личного бота, где можно работать прямо в одной ветке, например в master, и не заботиться о чистоте истории. Это простое решение, когда сложная организация веток просто не нужна.

Git Flow

Git Flow — одно из самых популярных решений, но лектор относится к нему сдержанно из-за сложности и медлительности. Основная ветка main хранит максимально стабильный код и почти не используется в активной разработке. Основная работа идет в develop, от которого отводятся feature-ветки для конкретных задач.

Релизы и hotfix в Git Flow

Когда develop достаточно стабилен, от него отводится release-ветка. Она вливается в main, после чего из main раскатывается production. Если нужно срочно исправить маленькую проблему, используется hotfix: ветка отводится прямо от main, вносится микроправка, и затем она сразу вмерживается обратно. Это удобно, когда нужно быстро поменять одну переменную или отключить проблемную фичу.

GitHub Flow

GitHub Flow проще: есть основная ветка main и feature-ветки. Разработчик создает фичу, реализует ее в отдельной ветке и потом вмерживает обратно. Лектор считает этот вариант самым простым и элегантным, особенно если хочется стартовать с минимальной сложности.

GitLab Flow

GitLab Flow — промежуточный вариант между Git Flow и GitHub Flow. Есть main с актуальным кодом, затем перед релизом отводится отдельная release-ветка,

после чего возможна еще одна стадия — production-ветка. Ветки для релиза сохраняются как отдельные сущности, и к ним можно вернуться, чтобы посмотреть, что именно было в конкретном релизе.

ГЛАВА 6

Trunk-based development и fork-in-flow

34:44 — 36:36

Лектор объясняет trunk-based development как подход, где разработка ведется почти в одной главной ветке с минимальным количеством короткоживущих веток и feature flags. Этот способ требует высокой дисциплины, но позволяет быстро тестировать и доставлять изменения. Также кратко рассматривается fork-in-flow как модель, характерная для open-source и pull request между разными репозиториями.

Идея trunk-based development

Trunk-based development — любимая стратегия лектора, но не все команды могут ее себе позволить. Смысл в том, что есть одна главная ветка, обычно не совсем main, а какая-то основная trunk-ветка, и разработка ведется прямо в ней. При этом допускаются только очень короткоживущие feature-ветки для мелких изменений.

Чем trunk-based отличается от других подходов

Главное отличие — минимизация числа веток и сокращение времени жизни дополнительных веток. Вместо сложной иерархии ветвления используются короткие, локальные изменения и быстрые вливания обратно в основную линию. Это снижает накладные расходы на ревью, merge и синхронизацию.

Feature flags как компенсация

Чтобы trunk-based работал, часто нужны feature flags, то есть флаги включения функций. Они позволяют держать в коде новую функциональность, но включать и выключать ее по необходимости. Это дает возможность быстро тестировать код без постоянного разветвления истории.

Практика в командах

Лектор приводит примеры: фронтендеры иногда используют trunk-based, но создают релизные ветки с номерами, например release 100, release 101 и так далее. По сути это одна длинная ветка, просто с формальным переименованием релизов. Бэкендеры чаще придерживаются develop и релизных веток. Выбор стратегии часто зависит от договоренности с ops-командой и того, какие процессы уже подстроены под GitOps.

Fork-in-flow

Fork-in-flow — это, по сути, модель работы в GitHub для публичных open source-репозиториев. Если проект чужой и публичный, нельзя просто взять ветку в том же репозитории и отправить merge request; обычно нужно форкнуть репозиторий и отправить pull request между двумя репозиториями. Это отдельная форма организации вклада в код.

ГЛАВА 7

Стратегии деплоя: от простого обновления к Blue-Green и Canary

39:30 — 48:38

Переход от CI/CD к самим стратегиям деплоя: лектор подчеркивает, что пайплайн не решает проблему простоя сам по себе. Разбираются простой деплой, частичное обновление, Blue-Green Deploy и Canary Deploy. Основная идея — уменьшить риск, дать возможность проверки на части инфраструктуры или части пользователей и упростить откат.

Что такое deployment strategy

Лектор подчеркивает, что CI/CD — это только пайплайн, конвейер доставки артефактов, но сами подходы к выкладке тоже важны. Как и с Git, где есть стратегия ветвления, здесь есть стратегия деплоя. Именно она помогает уменьшать простои и управлять рисками при выпуске версии.

Простой deploy и частичное обновление

Самый простой deploy — это обычное обновление всех сервисов примерно одновременно, как раньше до CI/CD. Это быстро, но рискованно: либо все взлетело, либо не взлетело, и откатывать не всегда удобно. Частичное обновление лучше: сначала обновить один сервис, потом другой, особенно если между ними есть зависимости, как в примере с Airflow и Spark.

Blue-Green Deploy

Blue-Green Deploy использует две идентичные среды, между которыми можно переключать трафик. Одна среда простаивает, вторая обслуживает пользователей; новую версию сначала разворачивают на простаивающем плече, проверяют и затем резко переводят на него весь трафик. Это удобно, быстро и хорошо тестируется.

Ограничения Blue-Green

Главный минус Blue-Green — стоимость: нужно держать две полноценные копии прода, и одна из них все время простаивает. Кроме того, переключение трафика, хотя и быстрое, не мгновенное. Если в момент переключения выполняется тяжелый запрос, он может быть затронут.

Canary Deploy

Canary Deploy не требует отдельного полноценного плеча: достаточно поднять вторую версию сервиса рядом и постепенно переводить на нее трафик. Например, можно сначала отправить 25% пользователей на новую версию и 75% оставить на старой, потом постепенно увеличивать долю новой версии. Это

дешевле, чем Blue-Green, быстрее откатывается и тестируется на реальных пользователях, но в малом объеме.

Сложность Canary

Минус Canary — сложность реализации. Система деплоя должна уметь перераспределять трафик и работать с несколькими копиями сервиса. Лектор приводит пример с Docker Compose и репликами: можно поднять сервис в нескольких экземплярах и направлять часть запросов на новую версию, а часть — на старую. Это уже почти искусственно созданный canary-режим.

ГЛАВА 8

A/B-тестирование, feature flags и rolling update

49:18 — 55:30

Обсуждаются A/B-тестирование и feature flags как более бизнес-ориентированные варианты частичного включения функций. Затем объясняется rolling update: реплики обновляются по одной, а трафик переводится только после успешных health checks и readiness-проб. Лектор отмечает, что такие подходы требуют стейтлесс-приложений, хорошего мониторинга и аккуратной оркестрации.

A/B-тестирование

A/B-тестирование по смыслу похоже на Canary, но больше относится к уровню бизнеса и экспериментов с фичами, чем к чисто техническому деплою. Здесь обе версии или обе ноды могут работать одновременно, а пользователи попадают на разные варианты. Разница с Canary в том, что здесь важен именно эксперимент над функциональностью.

Что проверяют в A/B

Лектор приводит пример с включением TLS на Postgres: на одном этапе проверяют, умеют ли сервисы работать через защищенный канал. Другой пример — включение фичи в приложении и наблюдение, пользуются ли ей вообще. Если фича не нужна, ее можно убрать и больше не использовать.

Feature flags

Feature flags — более точечный вариант той же идеи. Если нет возможности держать две копии сервиса или отдельные плечи, можно выкатывать один и тот же код, но управлять поведением через флаг внутри приложения. Например, интеграция с RabbitMQ может быть выключена по умолчанию и включена уже на проде.

Rolling update

Rolling update — постепенное обновление реплик, типичное для Kubernetes. Если сервис состоит из трех реплик, одна обновляется первой, две остаются старые; затем обновляется вторая, потом третья. Трафик при этом не переводится на новые реплики, пока весь набор не будет обновлен, а проверка состояния выполняется через health checks и probes.

Стейтлесс и риски

Для rolling update приложения должны быть stateless, а старая и новая версии должны сосуществовать без поломки базы данных и других зависимостей. Лектор отдельно поясняет, что если одна из реплик обновилась, нагрузка на нее не идет, пока все обновление не завершено и система деплоя не убедится, что

контейнер готов. Это делает rolling update более безопасным вариантом постепенного обновления.

ГЛАВА 9

Ответы на вопросы и практические рекомендации

55:30 — 57:39

В конце лекции лектор отвечает на вопросы о выборе стратегии ветвления и деплоя в рабочих проектах. Рекомендуется GitHub Flow как простой и гибкий стартовый вариант, а также поясняется разница между плечом в Blue-Green, отдельным инстансом и балансировкой трафика в Canary. Завершается лекция напоминанием о переносе следующего занятия.

Вопрос про выбор стратегии ветвления

На вопрос о том, какую стратегию ветвления стоит использовать в рабочих проектах с нуля, лектор отвечает, что GitHub Flow выглядит самым простым и практичным вариантом. В нем есть основная ветка, feature-ветки и простой merge обратно. Такой подход легче внедрить и потом при необходимости перейти на более сложные схемы.

Уточнение про Blue-Green

На вопрос о разнице между «плечом» в Blue-Green и просто отдельным инстансом лектор поясняет: в Blue-Green речь обычно идет о полноценной копии стенда, а не только одного сервиса. Если сильно упрощать, это почти два кубера с собственными балансировщиками, между которыми можно переключать трафик пользователей. Отдельный инстанс — это более локальная сущность.

Уточнение про балансировку

На вопрос о Docker Compose и репликах лектор подтверждает, что встроенный балансировщик там уже есть и может распределять трафик между репликами. При этом ничто не мешает использовать внешний балансировщик, например HAProxy. То есть способ распределения трафика можно реализовать как встроенными, так и внешними средствами.

Практический смысл A/B, feature flags и rolling update

Лектор еще раз связывает эти подходы с практикой: A/B и feature flags полезны для экспериментирования с фичами, а rolling update — для постепенной выкладки версий с проверкой готовности через пробы. На вопрос о том, что будет, если одна реплика в rolling update уже обслуживает трафик и падает под нагрузкой, объясняется, что обычно это решается оркестрацией и graceful shutdown, но теоретический риск есть.

Итог лекции

В конце лекции преподаватель подводит итог: были разобраны CI/CD, Git-ветвление и основные варианты деплоймента. Следующее занятие пропускается и будет наверстано позже, а далее тема продолжится уже на примерах в GitLab

и в контексте лабораторной работы. Он также напоминает про перенос занятия и завершает лекцию.

Ключевые цитаты

> «Continuous Integration и Continuous Delivery или Deployment.»

00:11 Это базовая расшифровка CI/CD, без которой дальше нельзя понять различие между этапами пайплайна.

> «Integration — это у нас интеграция нашей разработки в нашу инфру.»

00:11 Здесь лектор дает рабочее определение CI, связывая разработку с автоматической интеграцией в инфраструктуру.

> «Continuous Delivery предполагает только ручной деплой на продакшн.»

00:13 Эта фраза фиксирует ключевое различие между Delivery и Deployment.

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

00:14 Это второе ключевое определение, нужное для сравнения с Continuous Delivery.

> «Лучше уж побольше мелких коммитов, чем один большой.»

00:22 Это практическое правило, важное для читаемости истории и отката изменений.

> «Gitignore то же самое: записать, чтобы у нас не попало туда что-то лишнее, например секреты.»

00:23 Это важное следствие работы с Git: нужно исключать лишние и опасные файлы из репозитория.

> «main, который содержит самый актуальный код из возможных.»

00:26 Это центральная идея Git Flow, объясняющая роль основной ветки.

> «main у нас содержит актуальный код, но он не выкатывается в production.»

00:30 Это важное различие GitLab Flow: актуальный код хранится отдельно от ветки непосредственного релиза.

«Мы стараемся избежать большого количества веток, но добавляем множественные ключи, чтобы мы могли включать, выключать нужные фичи.»

00:34 Это суть trunk-based подхода и feature flags как альтернативы сложному ветвлению.

«CI/CD — это просто пайплайн, условно говоря, конвейер, который катит наши артефакты.»

00:39 Это емкая мемориальная формулировка, хорошо объясняющая роль CI/CD в доставке кода.

«Вторая у нас обычно среда, второе плечо, второе нода простаивает и просто ждет.»

00:43 Это основа понимания Blue-Green Deploy: одна среда активна, другая готовится к переключению.

«Направили 25 процентов пользователей на новую версию, а 75 оставили на старой.»

00:45 Это наглядное определение canary deploy через поэтапный перенос трафика.

«Здесь у нас разделение трафика частичное. Здесь оно полное.»

00:49 Эта фраза коротко фиксирует разницу между canary deployment и A/B-тестированием.

«Сервис из трех реплик: одну реплику обновляем, две остальные старые, и они все еще принимают на себя трафик.»

00:52 Это простое объяснение rolling update, полезное для запоминания механики обновления.

Глоссарий

CI/CD

Конвейер доставки кода, где CI отвечает за сборку и первичное тестирование, а CD — за дальнейшую доставку артефакта в тестовые среды и продакшн.

Continuous Integration

Часть CI/CD, связанная со сборкой кода и первичным тестированием результата.

Continuous Delivery

Вариант CD, где автоматизирован весь путь до финального ручного деплоя на продакшн.

Continuous Deployment

Вариант CD, где выкатка на продакшн тоже автоматизирована.

Артефакт

Результат сборки кода, который затем тестируется, хранится и доставляется по пайплайну.

Стейджинг

Промежуточная среда перед продакшном, где проверяют готовую версию.

Prod

Продакшн-среда, в которую доставляется конечный релиз.

Git Flow

Стратегия ветвления с main, develop, feature-, release- и hotfix-ветками.

GitHub Flow

Простая стратегия, где есть main и feature-ветки, а разработка и мердж идут через основную ветку.

GitLab Flow

Стратегия ветвления, в которой release-процесс организован через отдельные ветки для предпродакшна и продакшна.

Trunk-based development

Подход, при котором разработка ведется почти в одной главной ветке с минимальным числом короткоживущих веток.

Fork-in-flow

Модель, где изменения в open-source проектах вносятся через форк репозитория и pull request между репозиториями.

Hotfix

Срочное маленькое исправление, которое делается прямо от main без прохождения полного релизного цикла.

Blue-Green Deploy

Стратегия деплоя с двумя идентичными средами, между которыми переключают трафик после проверки новой версии.

Canary Deploy

Стратегия, где новая версия получает только часть трафика, а затем доля постепенно увеличивается.

A/B-тестирование

Подход, где разные пользователи получают разные варианты функциональности для сравнения и эксперимента.

Feature flag

Флаг в коде, который позволяет включать или выключать функциональность без отдельного разворачивания второй версии.

Rolling update

Постепенное обновление реплик, при котором старая и новая версии некоторое время существуют одновременно.

Health check

Проверка работоспособности обновленной реплики перед передачей ей трафика.

Readiness probe

Проверка готовности сервиса принимать трафик после обновления.

Вопросы для самопроверки

1. **Чем CI отличается от CD в составе CI/CD?**

   ОТВЕТ. CI — это сборка и первичное тестирование, CD — доставка артефакта дальше по средам, вплоть до prod.

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

   ОТВЕТ. В Continuous Delivery финальный деплой на прод выполняется вручную, а в Continuous Deployment — автоматически.

3. **Почему CI/CD повышает качество разработки?**

   ОТВЕТ. Потому что изменения многократно проверяются на разных стендах и процесс становится повторяемым.

4. **Зачем нужны feature branches в рабочих проектах?**

   ОТВЕТ. Чтобы изолировать изменения, упростить ревью, тестирование и безопасный merge в основную ветку.

5. **Когда уместен One Branch Flow?**

   ОТВЕТ. Для простых личных проектов или небольших задач, где строгая история и сложный релизный процесс не нужны.

6. **В чем проблема Git Flow по сравнению с GitHub Flow?**

   ОТВЕТ. Git Flow сложнее, медленнее и содержит больше сущностей и этапов, чем более простой GitHub Flow.

7. **Чем GitLab Flow отличается от GitHub Flow?**

   ОТВЕТ. В GitLab Flow добавляются промежуточные релизные ветки и более многоуровневая схема перед production.

8. **Что дает trunk-based development?**

   ОТВЕТ. Он уменьшает количество веток и ускоряет интеграцию, если команда умеет безопасно работать в главной ветке.

9. **В чем идея Blue-Green Deploy?**

   ОТВЕТ. Есть две одинаковые среды, новая версия проверяется на неактивной, затем трафик переключается целиком.

10. **Как работает Canary Deploy?**

    ОТВЕТ. Новая версия получает только часть трафика, и доля постепенно увеличивается при отсутствии проблем.

11. **Зачем нужен rolling update?**

    ОТВЕТ. Чтобы обновлять реплики постепенно и снижать риск массового отказа при релизе.

12. **Почему rolling update требует стейтлесс-приложения?**

    ОТВЕТ. Потому что старая и новая версии должны одновременно существовать без поломки состояния и логики данных.

Что изучить дополнительно

GitLab CI и построение пайплайнов в GitLab

Лектор прямо сказал, что реализация в GitLab будет на следующей лекции, а здесь тема раскрыта только на уровне общей схемы.

Infrastructure as Code

Лектор упомянул, что пайплайны и доставка описываются кодом, но не разбирал конкретные инструменты и практики.

Kubernetes probes и механика rolling update

Было кратко объяснено, что Kubernetes использует пробы для проверки готовности, но без детального устройства и примеров конфигурации.

Балансировка трафика в Blue-Green и Canary

Лектор объяснил принцип переключения и распределения нагрузки, но не углублялся в реализацию через ingress, proxy или service mesh.

Мониторинг и метрики для Canary и A/B

Было сказано, что нужны хорошие метрики, но не разобраны конкретные показатели, алерты и критерии остановки релиза.

Feature flags как инженерная и продуктовая практика

Флаги были показаны как способ включать функциональность, но без обсуждения управления флагами, жизненного цикла и технического долга.