CI/CD и роль DevOps

Предпосылки и причины возникновения CI/CD и DevOp

Проблемы "традиционной" разработки:

- **Разделение команд**

Разработка, тестирование и эксплуатация часто работали отдельно

Ситуация: Разработчики пишут код, передают его команде тестирования, те потом — в эксплуатацию. Никто не знает, что делает другая команда.

Пример: Разработчик написал новую фичу и отправил её на тестирование. Тестировщик говорит: «У меня не запускается». Оказалось, нужна была дополнительная библиотека, но разработчик об этом не сказал, потому что "у него всё работало".

- **Редкие релизы**

Новые версии ПО выходили редко, часто раз в несколько месяцев или даже лет

Ситуация: Релизы происходят раз в 3-6 месяцев. За это время скапливается куча изменений и багов, и релиз превращается в «мини-ад».

Пример: Финтех-компания обновляет своё мобильное приложение раз в полгода. Когда выходит новая версия:

- Пользователи жалуются на ошибки.
- Вылеты в приложении.
- Возврат к старой версии — почти невозможен.

Предпосылки и причины возникновения CI/CD и DevOp

- **Ошибки на проде**

Ошибки проявлялись только после выпуска

Ситуация: Тестирование происходит на одной машине, продакшен — на другой, с другими настройками.

Пример: Тесты проходят в тестовой среде (с одной БД). В проде — другая СУБД или конфигурация. В итоге:

- Запросы в базу начинают «падать».
- Приложение выдает 500-ошибки пользователям.

- **"У вас не работает, а у нас всё ок"**

Разные среды у разработчиков и у пользователей

Ситуация: Разработчик не может воспроизвести баг, потому что у него другая ОС, другое окружение, другой браузер.

Пример: Тестировщик сообщает о баге в Chrome 95, а у разработчика стоит Chrome 120. Версия Node.js тоже отличается. В итоге баг не воспроизводится — пока его не увидят на проде.

Какой вывод?

Требовалась новая философия и инструменты, чтобы:

- Быстро и часто выпускать новые версии.
- Минимизировать количество ошибок.
- Сблизить команды разработки и эксплуатации.

Кто такой DevOps-инженер?

DevOps — это культура и подход к разработке ПО, а **DevOps-инженер** — это специалист, который:

- Объединяет разработку (Dev) и эксплуатацию (Ops)
- Настраивает процессы CI/CD
- Автоматизирует развёртывание, тестирование, мониторинг.
- Работает с контейнерами (Docker), оркестрацией (Kubernetes), облаками (AWS, GCP, Azure)
- Поддерживает стабильную и быструю доставку кода в прод

Принципы DevOps

- **Автоматизируй всё, что можно**

Ручная работа — источник ошибок и потери времени. DevOps стремится к тому, чтобы процессы (тесты, сборка, запуск, деплой, проверка) выполнялись **автоматически**.

Пример: Вместо того чтобы вручную собирать приложение и загружать его на сервер — ты настраиваешь скрипт, который сам делает это каждый раз после коммита.

- **Непрерывная интеграция и доставка (CI/CD)**

Код проверяется и разворачивается **часто и быстро**, а не раз в месяц. Чем меньше изменений за раз — тем проще отследить баги.

Пример: Ты изменил 3 строки — пушишь в репозиторий → CI запускает тесты → если всё ок — код уходит на сервер в течение 5 минут.

Принципы DevOps

- **Dev и Ops работают вместе**

Раньше разработчики писали код, а админы его как-то запускали. В DevOps все работают **вместе**, чтобы продукт стабильно работал от идеи до продакшена.

Пример: Разработчик и DevOps-инженер вместе настраивают окружение, следят за логами и обсуждают, как оптимизировать нагрузку.

- **Наблюдение и обратная связь**

Важно **всегда знать**, что происходит с приложением: упало ли оно, сколько людей им пользуется, где тормозит.

Пример: Если приложение начало дольше отвечать — ты это увидишь в Grafana или получишь алерт на почту. Так можно быстро реагировать и чинить, пока пользователи не пожаловались.

- **Делай маленькими шагами**

Вместо огромных обновлений раз в год — делай небольшие изменения **регулярно**. Это проще протестировать, легче откатить, меньше рисков.

Пример: Вместо того чтобы сразу выкатывать весь редизайн сайта, ты внедряешь его по частям: сначала кнопки, потом шапку, потом карточки товаров.

Принципы DevOps

- **Fail fast — лучше пусть упадёт сейчас, чем потом**

Что значит: Ошибка на раннем этапе — не страшно. Главное — найти её **до продакшена**. Лучше пусть упадёт на тесте, чем на глазах у клиентов.

Пример: Разработчик сломал сборку — CI поймал это сразу. Баг не дойдёт до пользователей.

- **Infrastructure as Code (IaC)**

Серверы, базы данных, настройки — всё это теперь можно **описывать кодом**, как обычную программу. Это удобно, прозрачно и можно хранить в Git.

Пример: Ты создаёшь файл infrastructure.yml, где описано: «Создать 2 виртуальных сервера, установить туда Nginx и PostgreSQL». Потом запускаешь скрипт — и инфраструктура сама разворачивается.

Что такое CI/CD?

CI (Continuous Integration / Непрерывная интеграция) — это процесс, при котором разработчики **регулярно объединяют (merge/push)** свой код в общий репозиторий, а система автоматически проверяет, **работает ли код** и не ломает ли он остальное.

CD (Continuous Delivery / Deployment) — это процесс, при котором **собранный и протестированный код** автоматически разворачивается в разные среды: тестовую, staging, production — **без участия человека** или по нажатию кнопки.

Continuous Integration (CI)

Пример из практики:

1. Разработчик Вася добавил новый функционал (например, фильтрацию товаров в интернет-магазине).
2. Он делает **push** в репозиторий GitHub.
3. Запускается CI-пайплайн:

    - Проверка синтаксиса (линтер).
    - Юнит-тесты на функцию фильтрации.
    - Интеграционные тесты — например, фильтрация работает вместе с корзиной.
      Проверка сборки: собирается frontend и backend.

Если **все шаги проходят успешно** — коммит можно смело мержить в основную ветку.

Если тесты упали — код не сливается. Разработчику приходит уведомление (в Slack, почту, GitHub).

Инструменты для CI: Jenkins, GitLab CI/CD, GitHub Actions, CircleCI, Travis CI

Continuous Delivery (CD)

CI прошёл → начинается доставка в **тестовую или staging-среду**.

Пример из практики:

1. После успешного CI:
    - Код автоматически собирается в Docker-образ.
    - Этот образ заливается в Docker Registry.
    - Приложение разворачивается в **staging-окружении** на Kubernetes или виртуальной машине.

2. Там его проверяет QA-инженер, PM или даже заказчик.
3. Если всё работает — можно нажать кнопку “Deploy to Production”.

Инструменты для CD (delivery): Spinnaker, ArgoCD, Helm + Kubernetes, Ansible + Jenkins

Continuous Deployment (CD)

Разница с Delivery — **нет кнопки**. Всё автоматически разворачивается в production.

Пример из практики:

1. Код прошёл все этапы CI.
2. Образ загружен в Docker Registry.
3. Автоматически разворачивается на production (в облаке: AWS, GCP, Azure).
4. Через пару минут пользователи видят обновления.

Фишка: релизы могут происходить **несколько раз в день**. Это возможно, потому что:

- Все изменения маленькие.
- Каждый коммит проверен.
- Всё автоматизировано.

Разница CI и CD

	Continuous Integration (CI)	Continuous Delivery/Deployment (CD)
Что делает	Проверяет код при каждом коммите	Доставляет код в staging/production
Кто триггерит	Коммит или Pull Request	CI-пайплайн или кнопка/триггер
Автоматизация	Тесты, сборка, анализ кода	Доставка артефактов, деплой, миграции и т.д.
Цель	Быстрая обратная связь для разработчика	Быстрое и безопасное развертывание

Полный пример пайплайна CI/CD

Допустим, у нас есть Node.js-приложение с React frontend.

Шаги CI:

1. Git push → GitHub Actions запускает пайплайн.
2. Прогон линтеров (ESLint, Prettier).
3. Юнит-тесты (Jest, Mocha).
4. Сборка frontend (Webpack, Vite) и backend.
5. Интеграционные тесты (например, Cypress).

Шаги CD:

1. Сборка Docker-образа с backend + frontend.
2. Публикация образа в Docker Hub или GitLab Container Registry.
3. Развёртывание в staging:
    - Kubernetes + Helm.
    - Или просто Docker Compose на сервере.
4. Прогон e2e-тестов в staging.
5. Если всё ок — автоматический или ручной деплой в production.

Что нужно знать для работы с CI/CD?

Основы разработки:

- Git (основы, merge, pull request).
- Понимание жизненного цикла приложения.
- Основы тестирования.

Автоматизация:

- Bash / Shell скрипты.
- Jenkins / GitHub Actions / GitLab CI.
- Docker (контейнеризация).
- Kubernetes (по желанию, для сложных систем).

Работа с облаками:

- AWS, GCP, Azure.
- Terraform или Ansible (Infrastructure as Code).

Что нужно знать для работы с CI/CD?

Мониторинг и логирование:

- Prometheus, Grafana.
- ELK stack (Elasticsearch + Logstash + Kibana).
- Sentry, Datadog и др.

Заключение

CI/CD — это способ разработки, при котором:

- Всё автоматизировано.
- Ошибки ловятся рано.
- Продукт быстро доходит до пользователя. DevOps-инженер — связующее звено между разработчиками и продакшеном, которое делает всё это возможным.