DevOps/MLOps

Вы находитесь на финишной прямой курса «Data Engineer»: уже познакомились с различными инструментами работы с данными, коснулись темы ML и DL.

В этом лонгриде вы узнаете:

1. Что такое и зачем нужен DevOps.
2. Какие подходы существуют в DevOps.
3. Про монолитную и микросервисную архитектуру.
4. Какие существуют системы контроля версий.
5. Что такое контейнеризация и оркестраторы.

Что такое и зачем нужен DevOps

Как описывают DevOps разные источники:

1. Википедия: методология активного взаимодействия специалистов по разработке со специалистами по информационно-технологическому обслуживанию и взаимная интеграция их рабочих процессов для обеспечения качества продукта. Она предназначена для эффективной организации, создания и обновления программных продуктов и услуг. Основана на идее тесной взаимозависимости создания продукта и эксплуатации программного обеспечения, которая прививается команде как культура создания продукта.
2. Amazon: сочетание культурных принципов, подходов и средств, которое улучшает способность компаний создавать приложения и сервисы на высокой скорости. С DevOps разработка и оптимизация продуктов происходит быстрее, чем при использовании традиционных процессов работы над программным обеспечением и управления инфраструктурой. Благодаря такой скорости компании могут повысить уровень обслуживания клиентов и более эффективно конкурировать на рынке.
3. Microsoft: это сочетание разработки (Dev) и эксплуатации (Ops). Это объединение людей, процессов и технологий, позволяющее постоянно предоставлять преимущества клиентам. DevOps позволяет представителям ранее разрозненных подразделений — разработки, IT-операций, обеспечения качества и безопасности — координировать свои действия и совместно создавать более качественные и надёжные продукты.

Все три определения разные, в каждом выделены разные ключевые моменты. Тем не менее все они примерно об одном и том же: применение некоего принципа организации взаимодействия между участниками создания продукта обеспечивает этому продукту конкурентные преимущества и качество.

Основные подходы в DevOps

1. Continuous Integration (CI) направлена на то, чтобы код, который пишут разные люди, интегрировался в единую базу кода.

2. Автоматическое тестирование — практика, тесно связанная с CI. Виды автотестов:
    - Unit-тесты позволяют протестировать работу модуля в вакууме. Обычно пишутся короткими и могут запускаться регулярно.
    - Интеграционные тесты позволяют понять, как разные модули интегрируются между собой.
    - Тесты UI Automation позволяют проверить, насколько работа с UI соответствует требованиям, выставленным заказчиком. Продолжительность зависит от размера проекта. На большом проекте может составить от 2 до 5 часов и больше.

3. Непрерывное развёртывание позволяет залить чистый эталонный образ на новое окружение и посмотреть, к чему это приведёт.

4. Автоматическое восстановление — возврат на предыдущую версию приложения, если возникнут неполадки на актуальной. Пользователи продолжат работать с продуктом, а у вас появится возможность без спешки всё протестировать, пофиксить и залить новую версию.

5. Release Management совмещает непрерывное развёртывание и автоматическое восстановление, предполагает внедрение дополнительной абстракции (хранилища артефактов или галереи). После создания в ветке нового коммита приложение и всё важное из этого коммита пакуется в артефакт и сохраняется в хранилище, из которого его в любой момент можно извлечь. Артефакт последовательно разворачивается и тестируется на development, staging и production. Создание артефакта позволяет адресовать риски и в случае неудачного деплоя откатиться на предыдущую версию.

6. Infrastructure as Code позволяет описать инфраструктуру как код и при необходимости разворачивать окружение по кнопке.

7. Application performance monitoring предполагает периодическое (хотя бы раз в 2 недели) развёртывание приложения на отдельно стоящий сервер с чётко фиксированным процессором, оперативной памятью, дисками и т. д., прогон тестов производительности и отслеживание изменений от предыдущего спринта.

8. Configuration management — управление конфигурацией. На разном окружении может быть разная конфигурация, её настройку можно автоматизировать. Конфигурация всегда должна быть отделена от

приложения и управляться отдельно. Конфигурационные файлы лучше всего вынести за пределы приложения и контролировать через систему контроля версий. Если приложение представляет собой скомпилированный артефакт, находящийся на сервере приложений, все необходимые параметры этого приложения (конфиги, условия запуска и т. д.), лучше хранить в системе контроля версий, а не внутри приложения. Это позволит избежать пересборки приложения в случае изменения любого параметра. Достаточно будет внести команду изменения параметров в репозиторий с инструментом.

9. CI/CD объединяет Application performance monitoring и Configuration management. Его мы разберём на следующих уроках.

10. Continuous training — один из вариантов деплоя модели в production. После разработки пайплайна обучения модели, тренировки базовой модели и деплоя в production проводится перетренировка модели.

Монолитная и микросервисная архитектура

Монолитное приложение (монолит) — приложение, доставляемое через единое развёртывание.

В микросервисной архитектуре слабо связанные сервисы взаимодействуют друг с другом для выполнения задач, которые относятся к их бизнес-возможностям.

В отличие от монолита, в микросервисах есть несколько единиц развёртывания. Каждый сервис развёртывается самостоятельно.

Достоинства и недостатки монолитной архитектуры

Достоинства:

- Простота разработки — IDE и другие инструменты разработки сосредоточены на построении единого приложения.
- Лёгкость внесения радикальных изменений — вы можете поменять код и структуру базы данных, а затем собрать и развернуть полученный результат.
- Простота тестирования — разработчики пишут сквозные тесты для запуска приложения, обращения к REST API и проверки пользовательского интерфейса с помощью Selenium.
- Простота развёртывания — разработчику достаточно скопировать WAR-файл на сервер с установленной копией Tomcat.

Недостатки:

- Высокая сложность. По мере развития приложение становится слишком большим, исправление ошибок и реализация новых возможностей усложняется. Усугубляет проблему то, что сложность, и так чрезмерная, обычно повышается экспоненциально. Если кодовая база плохо поддаётся пониманию, разработчик не сможет внести изменения подходящим образом. Каждое изменение усложняет код и делает его менее понятным.

- Медленная разработка. Помимо борьбы с чрезмерной сложностью, разработчикам приходится иметь дело с замедлением ежедневных технических задач. Большое приложение перегружает и замедляет их IDE, сборка кода занимает много времени, приложение долго запускается. В итоге затягивается цикл написания, сборки, запуска и тестирования кода, что плохо сказывается на продуктивности в целом.

- Большой срок от внесения изменений до развёртывания — внесение изменений, тестирование и исправление багов занимает много времени.

- Трудности с масштабированием. Разные программные компоненты имеют разные требования к ресурсам. Например, одни компоненты требуют высокой скорости чтения и записи на диск, другие сильно нагружают CPU/GPU. Компании приходится идти на компромисс при выборе серверной конфигурации.

- Проблемы с надёжностью — из-за размеров приложение сложно протестировать должным образом и локализовать ошибку.

- Высокая зависимость от стека технологий — сложный переход на новые фреймворки и языки программирования. Переписывать монолитное приложение чрезвычайно дорого и рискованно. Программистам приходится работать инструментами, выбранными при запуске проекта, и поддерживать код, написанный с помощью устаревших средств.

Достоинства и недостатки микросервисной архитектуры

Достоинства:

- Возможность непрерывной доставки и развёртывания — достигается необходимый уровень тестирования, т. к. небольшой сервис покрыть тестами и проверить намного проще. Из-за независимости сервисы могут быть развёрнуты отдельно, не нужно ждать всех изменений.

- Простота обслуживания сервиса — сервис небольшого размера легко разрабатывать и дебажить.

- Независимое масштабирование сервисов — при возрастании нагрузки на определённый компонент бизнес-логики его можно масштабировать.

- Автономность команд разработчиков. Команды, которые реализуют определённую бизнес-логику, разрабатывают и развёртывают свой сервис вне зависимости от других команд.

- Возможность экспериментировать и внедрять новые технологии.

- Лучше изолированы неполадки — проблемы в сервисе отразятся на работе только этого сервиса, не затронув остальные. Но если сервис состоит из цепочки с интеграциями, то весь функционал может стать недоступным, кроме того, который несёт каждый сервис по отдельности. Это стоит учитывать при планировании архитектуры микросервиса и его связей.

Недостатки:

- Сложность подбора подходящего набора сервисов — отсутствует конкретный, хорошо описанный алгоритм разбиения системы на микросервисы. В случае неправильного разделения системы получится распределённый монолит — набор связанных между собой сервисов, которые необходимо развёртывать вместе. Распределённому монолиту присущи недостатки как монолитной, так и микросервисной архитектуры.

- Затруднения при разработке, тестировании и развёртывании. Сервисы должны использовать механизм межпроцессного взаимодействия. Код должен уметь справляться с частичными сбоями и быть готовым к недоступности или высокой латентности удалённого сервиса. Реализация сценариев, охватывающих несколько сервисов, требует применения незнакомых технологий. Каждый сервис имеет собственную базу данных, что затрудняет реализацию комбинированных транзакций и запросов. Затруднено написание сквозных тестов. Вдобавок микросервисная архитектура существенно усложняет администрирование. В промышленной среде приходится иметь дело с множеством экземпляров разнородных сервисов.

- Принять решение о переходе на микросервисную архитектуру трудно. Сначала нужно определиться, на каком этапе жизненного цикла приложения следует это сделать. Чаще всего проблема связана с совокупными большими трудозатратами в человеко-часах, поэтому не каждый монолит переносят на микросервисы.

Какая архитектура лучше?

Микросервисная архитектура используется чаще. Во время разработки первой версии вы ещё не сталкиваетесь с проблемами, которые она решает. Более того, применение сложного распределённого метода проектирования замедлит разработку. Для стартапов, которым важно быстро развивать свою бизнес-модель и сопутствующее приложение, это может вылиться в непростую дилемму. Использование микросервисной архитектуры усложняет выпуск

начальных версий. Стартапам почти всегда лучше начинать с монолитного приложения.

Однако со временем возникает другая проблема: как справиться с возрастающей сложностью. Это подходящий момент для того, чтобы разбить приложение на микросервисы с разными функциями.

Микросервисы — это хорошо, но не забывайте про коробочные решения, с которыми сталкиваются различные компании. Большинство вендоров предлагают коробки, являющиеся монолитом. Это надо учитывать, если вы решили идти не по пути внутренней разработки, а выбрать продукты под ваши нужды на рынке.

Важно также учесть, что микросервисы должны исходить и жить в парадигме «Один сервис — одна проблема», как это следует из принципа Unix «Одна программа — одно решение». Если ваш сервис начнёт обрастать функционалом, то он постепенно станет монолитом. Поэтому подобные вещи лучше контролировать на уровне архитектуры, когда вы ходите развить или добавить новый функционал. Лучше создать для нового функционала сервис, чем добавлять его к существующему.

Системы контроля версий

Системы контроля версий подразделяются в зависимости от предмета классификации:

- версионирование кода — Git, Mercurial;
- версионирование артефактов — Artifactory;
- версионирование моделей — MLflow;
- версионирование данных — DVS.

Контейнеризация и оркестраторы

В работе мы сталкиваемся с различными языками программирования, библиотеками, технологиями, устройствами. Все они имеют определённые зависимости, которые часто несовместимы друг с другом. Чтобы решить проблему управления зависимостями, придумали технологию виртуализации. Одна из её разновидностей — контейнеризация.

Контейнеризация — метод виртуализации, при котором ядро операционной системы поддерживает несколько изолированных экземпляров пространства пользователя вместо одного.

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. Ему будет посвящён отдельный урок.

Кластер — разновидность параллельной или распределённой системы, которая используется как единый ресурс и состоит из нескольких виртуальных или hardware-сущностей, распределённых согласно DRP и объединённых в единый комплекс для отказоустойчивости.

Оркестрация — управление оркестром сервисов внутри единой информационной системы. Выбор оркестратора зависит от задачи. Самые популярные оркестраторы Kubernetes и Docker Compose мы разберём на следующих лекциях.

No Silver Bullet

Фредери́к Брукс в книге «Мифический человеко-месяц, или Как создаются программные системы» писал: *«Нет ни одного открытия ни в технологии, ни в методах управления, одно только использование которого обещало бы в течение ближайшего десятилетия на порядок повысить производительность, надёжность, простоту разработки программного обеспечения».*

Работая в компании, стартапе, производя какой-либо продукт, нужно понимать в каком контексте вы находитесь. То, что работает у других, не факт, что будет работать у вас. А то, что вы применяли в прошлых проектах, возможно, не подойдёт для будущих.