KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект >1 урок > Инфраструктура и процессы в современных бекендах

> Оглавление

> Вступление

Мы начинаем модуль **жизненного цикла ML** — модуль об обучении моделей, доведении их до прода и последующей поддержке. Даже если вы только обучаете модели и не занимаетесь ETL процессами, сопряжёнными с обучением, выводом моделей в прод и их поддержкой (и не хотите этим заниматься), вам всё равно придётся думать как инженеру, ответственному за эти процессы, ведь модель в конечном итоге должна работать не в тетрадке (хотя в некоторых компаниях предлагают деплоить именно Jupyter-тетрадки), а в существующей инфраструктуре с существующими бэкендами.

Например, вы делаете модель для какой-то системы реального времени и использовали для обучения данные из SQL базы данных. И запросы, добывающие эти данные, могут отрабатывать до 10 секунд, что неприемлемо для реального времени. Конечно, можно создать инфраструктуру, ETL процессы, которые вместе смогут заменить эти 10-секундные запросы, но тут возникает вопрос: "А кто эту инфраструктуру или процессы будет проектировать и настраивать?" Даже если не вы, вам всё равно необходимо

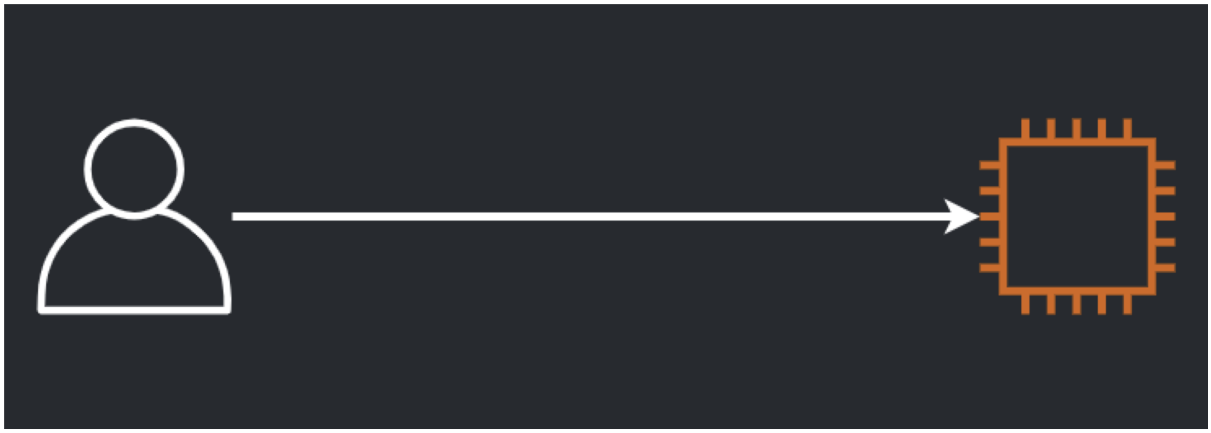
иметь хотя бы минимальное представление о том, как это работает, чтобы поставить техническое задание.

Другой пример: вы строите модель рекомендаций для сайта по сессиям пользователей, но сессии и клики внутри сессий, как события на бэкенде, попадают в аналитические базы через длинный путь стримингов/пайплайнов, которые могут занимать несколько часов (возможно даже дней). В результате посчитанные на сессии рекомендации протухают к следующим сессиям пользователя. Конечно, тут можно отказаться от user-based рекомендаций и перейти на item-based рекомендацию, которая протухает гораздо медленнее, но вдруг у вас в компании принят кодекс "бусидо", и вам придётся искать способ встраиваться в инфраструктуру, чтобы получать данные раньше, чем через N десятков минут, чтобы отдавать рекомендации по сессиям. Если же с архитектурой и инфраструктурой есть возможность ограничиться общим знанием и пониманием, то вот с процессами и практиками разработки лучше знакомиться значительно ближе, потому что ML — это та же разработка софта, только с ещё большей головной болью, и, скорее всего, вы уже развиваете ML в каком-то живом продукте, в котором есть бэкенд и какие-то процессы.

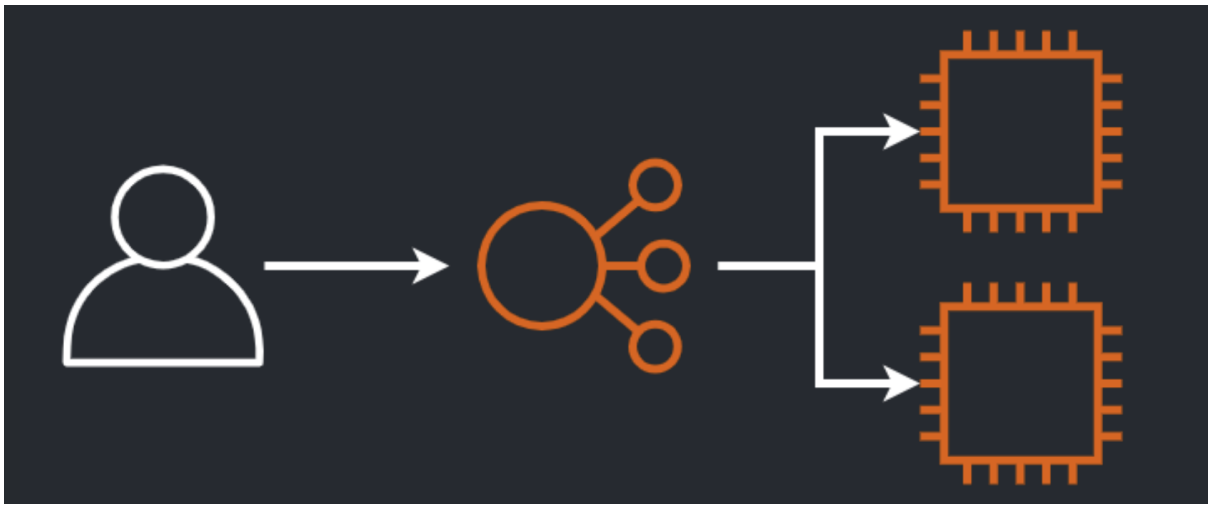
Поэтому сначала предлагаем пройтись по базовым аспектам современных бэкендов, по образу и подобию которых лучше всего проектировать свои приложения, и узнать о процессах, которые являются устоявшимися стандартами в разработке. Если к тому моменту вы не примете решение перекатиться в бэкенд или девопс, то обсудим, что из рассмотренного можно использовать в ML-системах, а чего явно не хватает.

> Современные бэкенды

Рассмотрим среднестатистический современный бэкенд, который, скорее всего, будет в вашем проекте. У вас есть приложение — веб-сервис, который принимает запросы и отдаёт ответы. Выглядит не очень современно, а главное — ненадёжно. Любая неполадка с приложением выводит весь проект из строя.

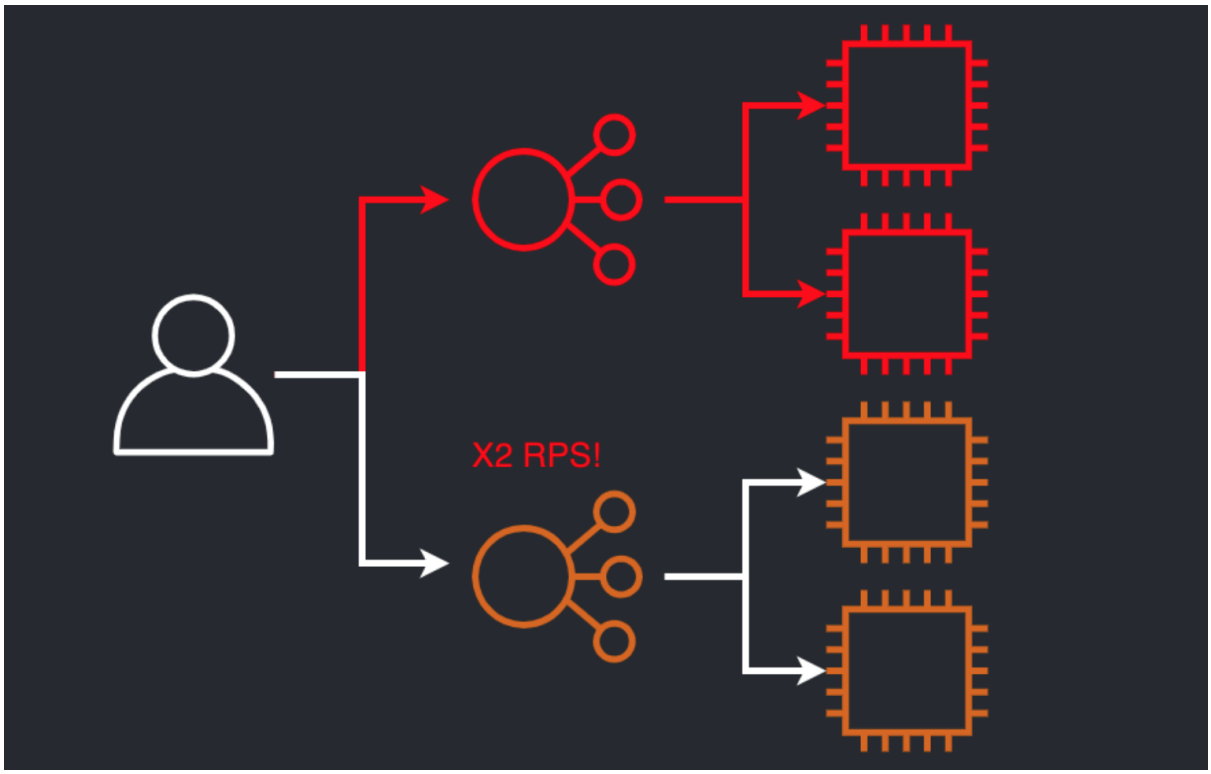


Поэтому вы добавляете ещё один точно такой же бэкенд, чтобы в случае выхода из строя одного работал другой. Перед ними вы поставили балансировщик, который балансирует трафик между двумя репликами бэкенда. Трафик распределяется без учёта пользователей или сессии, т.е. два запроса от одного пользователя могут попасть на разные реплики, что совсем не страшно, поскольку они одинаковы. У вас может быть кейс, в котором вы захотите, чтобы в рамках одной сессии вы попадали не только на одну конкретную реплику. Существует механизм, который позволяет направлять запросы сессии только на одну реплику — он называется sticky sessions, но сейчас его стараются не использовать, чтобы можно было без затруднений реализовать схему на рисунке. Её плюсы мы обсудим позже.

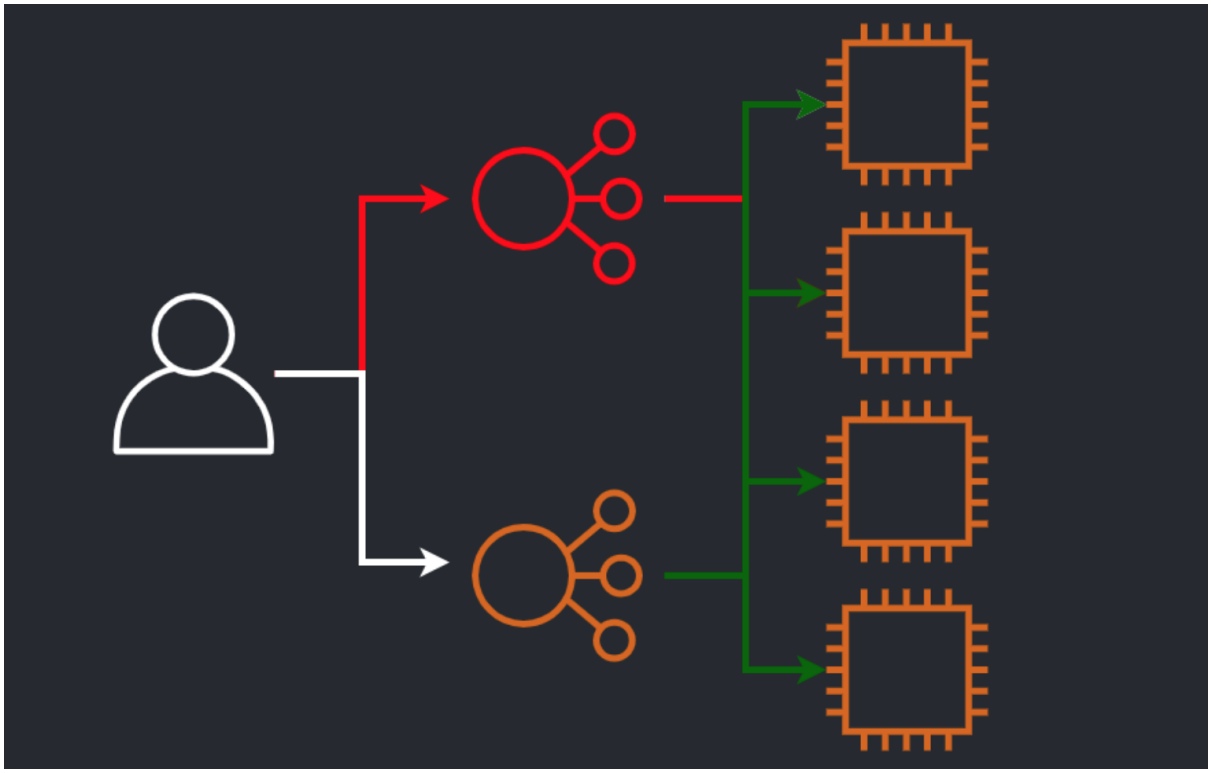


Две реплики уже хорошо, но нагрузка растёт, и их не хватает. Давайте сделаем четыре, а также вспомним, что балансировщик тоже может отказать. Поэтому поставим второй, и наши пользователи будут случайным образом попадать на один из двух балансировщиков. Теперь, если отказывает один из инстансов

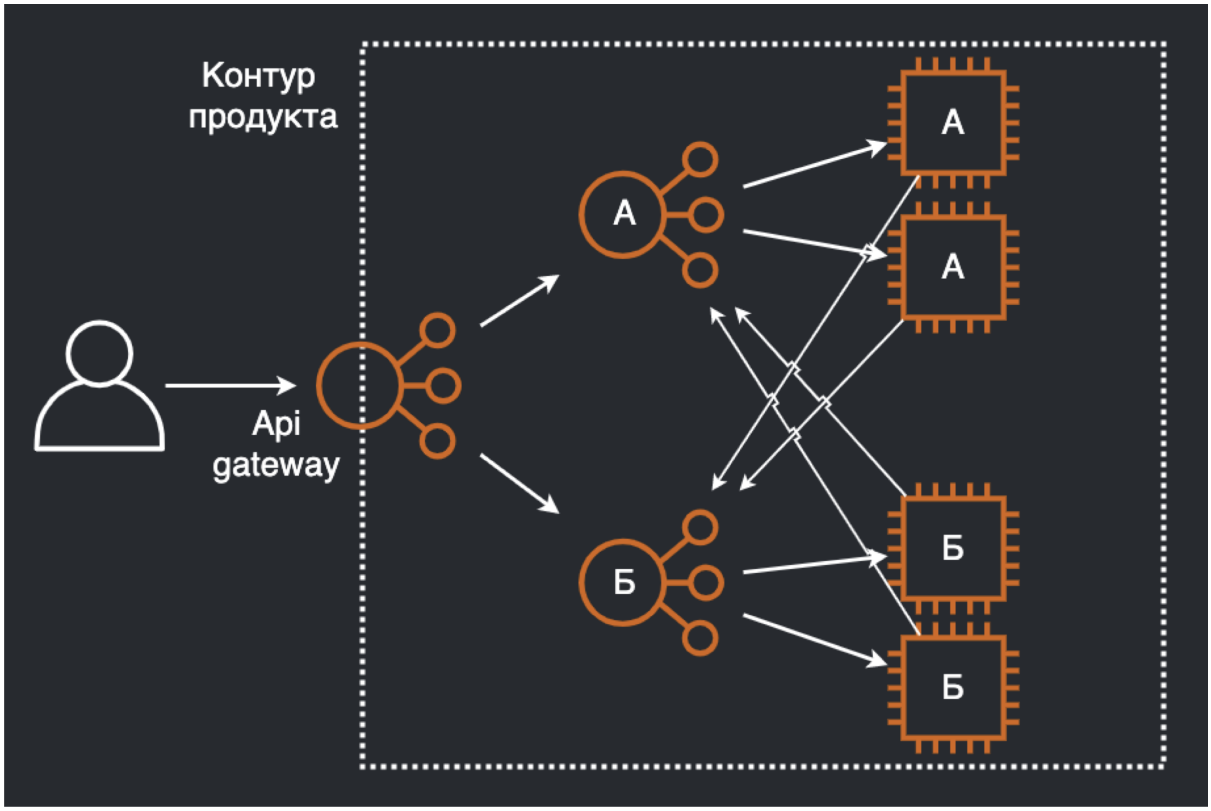
бэкенда, 75% мощностей будут держать всю нагрузку. Мы это учли, поэтому 75% процентов пока хватает. А теперь давайте представим, что отказала не реплика бэкенда, а балансировщик.



Тогда на второй балансировщик полетит двойная нагрузка, которую он скорее всего выдержит, а вот две реплики бэкенда могут не выдержать двойной нагрузки, что может привести к их падению. Поэтому мы сделаем разумнее — вместо того чтобы делить инстансы на две изолированные группы, мы настроим их получать трафик от каждого из балансировщиков. Таким образом, если мы теряем балансировщик, у нас остаётся 100% мощностей. Такая архитектура уже довольно устойчивая.

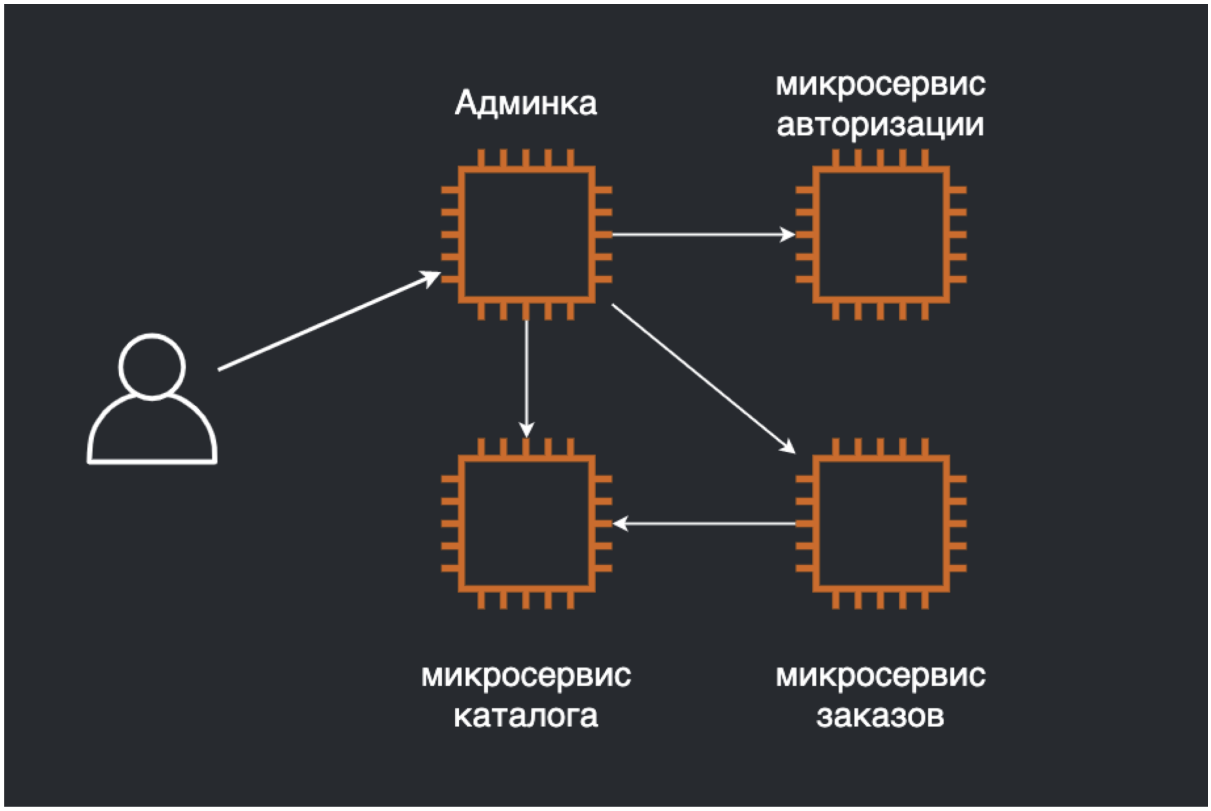


Наш проект развивается, рядом начинают вырастать другие микросервисы. На следующей схеме с двумя микросервисами А и Б продукт находится уже в более зрелом состоянии — клиент попадает в микросервис не напрямую и даже не через его балансировщик, а через какой-то другой интерфейс доступа к вашим сервисами, например API Gateway шлюз, Kafka или корпоративный балансировщик. Микросервисы вашего продукта общаются друг с другом через балансировщики: например, если реплике микросервиса А нужно обратиться в микросервис Б, она кинет запрос на балансировщик, который в свою очередь отправит запрос на одну из реплик Б.



> Микросервисы

Поскольку мы уже упомянули микросервисы, то пару слов нужно сказать и о них: сейчас микросервисы приплетают при любом удобном случае, поэтому вы наверняка уже всё про них знаете, но давайте на всякий случай проговорим это ещё раз.



Микросервисная архитектура — это способ декомпозиции кода, но не на уровне исходного кода (классов и функций), а на уровне инфраструктуры, т.е. когда разные куски кода выносятся в разные самостоятельные приложения. Например, вы можете сделать микросервисы, отвечающие за сохранение товаров, за продуктовую корзину, прайсинг и авторизацию. То есть разделение происходит как по бизнес-функционалу, так и по техническому функционалу. На эту архитектуру переходят, потому что она упрощает жизнь по сравнению с монолитом, когда вся логика в одном большом приложении.

Преимущества:

1. Разные стеки (клиентские — nodejs, обработка данных — python);
2. Возможна параллельная разработка;
3. Тестирование легче (меньше логики — меньше тест-кейсов);
4. Независимая развёртка (обновление логики работы сервиса затрагивает только его);
5. Быстрый запуск и быстрое завершение работы;

6. Когнитивные плюсы (все плюсы, вытекающие из небольшого размера сервиса).

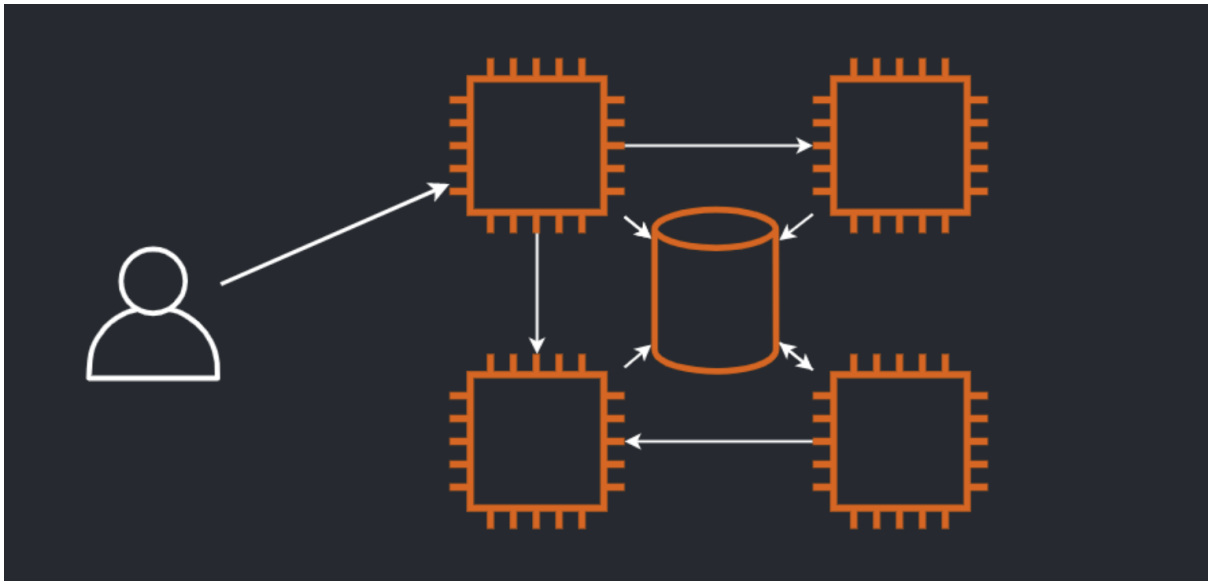
Недостатки:

1. Сложнее переиспользовать код (нужно дублировать изменения во всех проектах);
2. Контракты взаимодействия (они нужны для общения между сервисами — подобие REST API);
3. Race Condition (сервис-обработчик не успел обработать документ, а клиентский микросервис уже отдал список обработанных);
4. Нагрузка на инфраструктуру (больше приложений, а значит, приходится много оркестрировать);
5. Усложнённая развёртка (например, перед развёрткой необходимо убедиться, что сопряжённые микросервисы доступны);
6. Версионность (микросервисы могут выпускаться с разной частотой, поэтому необходимо обеспечивать обратную совместимость);
7. Сетевой трафик (его много, но это вряд ли станет проблемой в ближайшее время).

Микросервисная архитектура интуитивна, и ей просто следовать. Достаточно соответствующим образом спроектировать приложения и следить, чтобы в них не появлялись антипаттерны, самым ярким из которых является централизация. Мы говорим об архитектуре, которая по природе своей децентрализованная, поэтому если у вас появляется какой-то самый главный микросервис, в который ходят все остальные микросервисы, то лучше подумать о других вариантах. Сюда же относится аккумулирование состояний — в сервисах лучше вообще не держать внутреннего состояния. Про другие антипаттерны можно почитать тут.

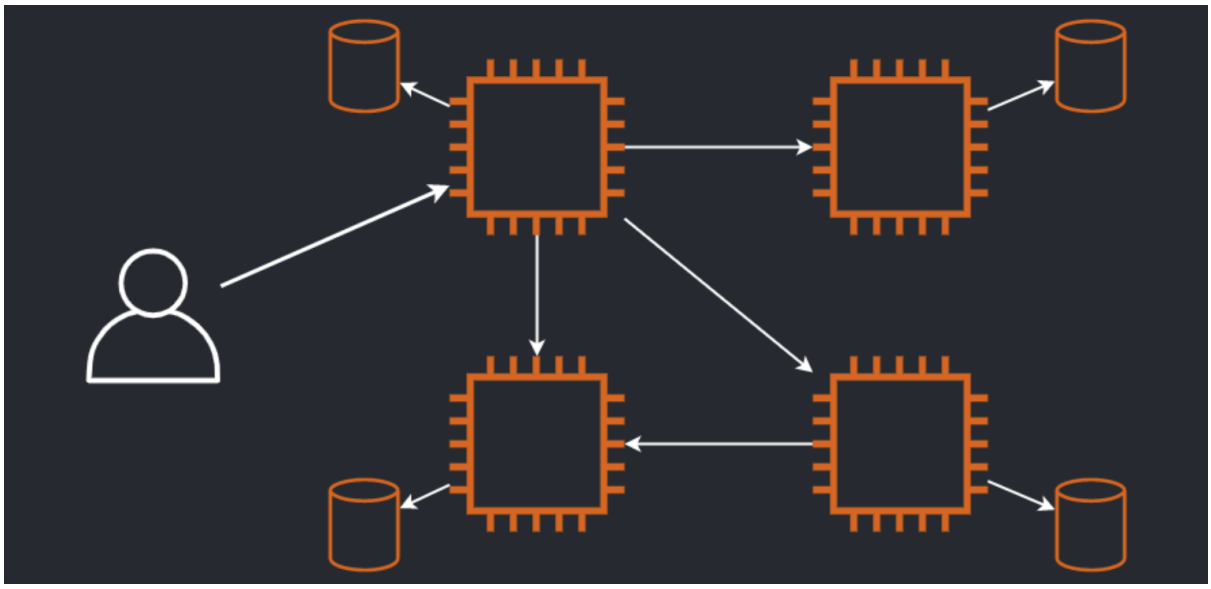
> Базы данных

Теперь, когда мы немного разобрались с архитектурой, поговорим о базах данных.



Сейчас для проектов чаще всего выбирается PostgreSQL, поэтому по умолчанию будем иметь в виду её. Итак, продукт обрастает микросервисами, но что происходит с базой? Неужели одна база будет обслуживать все микросервисы? Конечно, это возможно, но лучше так не делать. Существует паттерн проектирования, согласно которому каждый компонент может писать только в один источник. Согласно этому паттерну, каждому пишущему микросервису требуется своя база.

> Database per Service



Здесь БД — это не обязательно физически отдельный процесс PostgreSQL со своей файловой системой или вообще на отдельной машине, это логическая

база внутри одной физической БД. Однако в идеале лучше, чтобы это была полностью отдельная база.

На самом деле паттерн можно расширить и до чтения: каждый компонент пишет и читает из одного источника. Допустим, микросервису, отвечающему за личный кабинет продавца, нужно получать товары. Эти товары он получит не из базы товаров напрямую, а сходит в микросервис, отвечающий за каталог, и получит данные по API. Также, если администратору торгового центра нужно заблокировать продавца, он пойдёт по API в микросервис личного кабинета продавца, а не полезет сам в базу деактивировать продавцов.

Это паттерн — ему можно не следовать, но всё же приведём некоторые аргументы в его пользу:

1. Простое обновление схем. Если несколько микросервисов пользуются одними и теми же таблицами, то обновлять схему таблиц будет сложно, т.к. обновления для нескольких сервисов должны выкатиться одновременно с обновлением схемы данных, ведь сервисы, которые не успеют обновиться или наоборот обновятся раньше, чем база, будут недоступны, поскольку модели данных в бэкенде не будут совпадать с данными в базе. А организовать такое одновременное обновление серивсов и базы не так уж и просто. Есть ещё один вариант развития событий — в коде сервисов можно поддерживать возможность работы как со старой, так и с новой версиями базы, но это довольно трудозатратно. А если у каждого сервиса будет своя база, то не будет общих таблиц, и обновлять схему данных будет очень просто.
2. Хорошая сочетаемость с микросервисным подходом. Суть микросервисов как раз в том, чтобы делить логику, но если микросервис будет совершать действия напрямую со всеми базами, то придётся реализовать и логику для обработки этих данных, что нивелирует плюсы микросервисного подхода.
3. Диверсификация рисков падения. Если база данных упала или на неё накатили неправильные миграции, тогда сервис, потеряв возможность читать из базы, может иметь какую-то логику ответа на запросы, даже если недоступен его основный источник. В том случае, когда БД была одна на всех и упала, скорее всего уже ничего не поделать — вся система будет недоступна, пока БД не поднимется (рекомендуется также почитать про такие свойства систем, как fail-over или fault-tolerance).
4. Выбор БД для каждого сервиса. Разным сервисам может подходить разная база — например, админка на PostgreSQL, а микросервис заказов на

MongoDB.

5. Контроль нагрузки БД. Нагрузка на источник идёт только от сервиса, и сервис её может контролировать, если видит, что нагрузка очень большая. Сервис мы почти всегда можем масштабировать горизонтально — нужно сделать больше реплик, а вот с масштабированием БД всё сложнее. Если характер нагрузки на базу сложный (много разных транзакций update'ов и insert'ов), то дефолтный контейнер Postgres'а может на 50 запросах в секунду уже серьёзно падать в производительности. Поэтому существуют техники настройки базы. Одна из популярных — создание 3 реплик базы master-slave-slave. Эти реплики не равноправны, как реплики бэкенда. Запись и обновление происходит в master, а чтение — из slave. Изменения из master синхронизируются в slave'ах. Кстати, это также добавляет к отказоустойчивости, т.к. если master отвалится, то один из slave занимает место master'а. Безусловно, есть способы оптимизировать БД, но это труднее, чем просто создать реплики в бэкенде — лучше всего спроектировать систему так, чтобы эту оптимизацию делать не пришлось. А если, согласно паттерну, у каждого микросервиса будет своя база, то нагрузка будет распределяться по базам, и потребности в оптимизации будет существенно меньше. К тому же будет гораздо проще понимать характер нагрузки на каждую конкретную базу и, если что, адаптировать своё приложение под неё.

Конечно, вместе с удобствами приходят и неудобства:

1. Невозможность делать некоторые запросы. Теперь из-за того, что данные хранятся в разных базах, вы, вероятно, не сможете делать некоторые SQL запросы, потому что нужная для join таблица хранится в другой базе. Вам придётся спроектировать endpoint в соответствующем микросервисе, чтобы вытаскивать эту информацию из соответствующего сервиса.
2. Синхронизация таблиц. Если же всё-таки endpoint'ом не обойтись и нужно, чтобы в одной базе была, например, таблица из другой, то придётся либо писать скрипты для копирования таблицы, либо для синхронизации нужной таблицы из одной базы в другую пользоваться такими инструментами, как dblink.

> Сетевое взаимодействие

Теперь перейдём к другому аспекту бэкенда — сетевому взаимодействию.

Микросервисы используют сетевое взаимодействие. Почти всегда это REST API, что подразумевает использование HTTP или HTTPS. В принципе, REST можно реализовать и на других протоколах, потому что REST — это паттерн (его мы затронем чуть позже). В основном все используют HTTP 1-й версии, но если требуется эффективность, то можно использовать HTTP/2 или GRPC, который является надстройкой над HTTP/2.

- HTTP/1.1 (REST API) — простой, и, как правило, поддерживается всем, концепция 1 запрос — 1 ответ.
- HTTP/2 — близок к REST API, позволяет отправлять несколько запросов на 1 ответ.
- GRPC — построен на HTTP/2, используется строгая типизация, предоставляет инструменты для кастомизации протокола.

Кстати, вы вероятно знаете, что HTTP — высокоуровневый сетевой протокол над TCP, из-за чего в HTTP присутствует некоторый overhead, поэтому сорвиголовы иногда используют прям TCP сокеты для связывания сервисов (без HTTP).

> Наблюдаемость

Никакая система не может эффективно эксплуатироваться, если она ненаблюдаема. Наблюдаемость — это важное свойство, которое помогает и при решении инцидентов, и при планировании ресурсов. Для обеспечения этого качества системы есть принципиально 2 разных подхода, дополняющих друг друга:

- Сбор метрик
- Сбор логов

> Метрики

- Количество запросов в единицу времени, загрузка процессора, количество свободного места на диске, задержка обработки запроса и пр.
- На определенные метрики (их пороговые значения) создаются alert'ы (сигналы тревоги). Для этого уже есть системы, использование которых принято считать стандартом индустрии, например Prometheus.

> Логи (текстовые сообщения)

- Логи приложений агрегируются (например с помощью Sentry).
- Агрегированные логи визуализируются (например ElasticSearch в с панелью Kibana).

Согласно 12-Factor App (перечню лучших практик для проектирования и реализации микросервисов), мы просто пишем логи в потоки stdout/stderr, а дальше переслать их в хранилище — это уже дело инфраструктурной платформы.

Sentry даёт готовую аналитику по различным типам исключений, чего нет у ElasticSearch в базе, хотя это на самом деле можно реализовать самостоятельно. Также нельзя не затронуть тему сбора телеметрии — трейсингов (tracings). Это опять же требует инструментария приложения и позволяет собирать дерево вызовов между различными микросервисами. Это очень мощный инструмент, который появился как ответ на возросшую сложность взаимодействий между микросервисами. Примером такого решения является Jaeger.

> 12-Factor Apps

На прошлом шаге был упомянут термин 12-factor apps. Дело в том, что сейчас всё ПО разрабатывается в основном как веб-сервисы, т.е. как SaaS, и очевидно, что в индустрии накопился опыт, на основе которого были сформулированы советы по проектированию и развёртке приложений. В нашей лекции некоторые из 12 принципов уже встречались, поэтому сейчас мы кратко рассмотрим ещё не упомянутые пункты.

> Кодовая база

Суть в том, чтобы на один репозиторий иметь только одно приложение. Если у вас есть веб-сервис, использующий TensorFlow и PyTorch, но, допустим, у них одинаковая предобработка данных, то, возможно, вы захотите держать код обоих сервисов в одном репозитории, чтобы не дублировать код предобработки. Это неправильно — в следующем уроке мы поймём, почему. Сейчас отметим основные варианты:

- Дублировать код предобработки (но дублирование, как мы уже знаем, нежелательно).

- Выделить код предобработки в отдельный репозиторий и использовать его в проектах с PyTorch и TensorFlow как подмодуль в Git. Наш преподаватель и автор текущего блока Владислав Ладенков советует прибегать к подмодулям (submodules) в последнюю очередь, потому что с ними не очень удобно работать и люди постоянно допускают ошибки. Также возможны и технические трудности — например, ваша CI (Continuous Integration) система скачает репозиторий проекта, а submodule, который в нём используется, будет проигнорирован.
- Выделить код предобработки в отдельный репозиторий и сделать из него установочный пакет, чтобы приложения устанавливали его как зависимость.
- Выделить предобработку в микросервис.

> Зависимости

Нужно объявлять все зависимости и объявлять их явно вместе с версиями. Иначе при переносе приложения на другие окружения может случиться так, что найденная версия какого-то пакета не подойдёт, приложение не запустится и вам придётся искать причину.

Также эти зависимости принято изолировать: вы наверняка пользовались virtualenv'ом или conda. Ваши веб-сервисы тоже должны быть запущены в виртуальном окружении. Но так как вы, скорее всего, будете использовать контейнеры, то изоляция вас не должна тревожить, ведь контейнеры по своей сути изолируют зависимости.

> Отделение кода от конфигурации

Если кратко, то лучший способ хранить конфигурации — это переменные окружения. Подробнее рассмотрим их чуть позже.

> Все сервисы как подключаемые службы

Этот пункт перекликается с предыдущим. Все базы данных, API и т.д. (локальные или внешние) должны быть легкозаменяемыми, т.е. не должно быть хардкода.

> Строгое разделение стадий (сборка, релиз, выполнение)

Советуют строго разделять эти стадии, однако как в нашем проекте, так и в некоторых других проектах стадии релиза и выполнения могут сливаться в одну стадию.

> Приложения как StateLess процессы

Это хорошо объясняется здесь. Добавим только, что наши микросервисы в нескольких инстансах за nginx, которые мы наблюдали на схемах, фактически и есть приложение в нескольких процессах.

> Привязка портов

В домашних заданиях мы создавали Flask-сервисы и публиковали их на localhost вместе с портом. Но это не единственный способ — есть вариант с веб-серверами, которые подгружают ваше приложение как плагин и публикуют сервис уже по своим правилам.

Вы делали всё правильно. Владислав Ладенков уверяет, что вообще ни разу не видел в компаниях реализацию 2-го подхода — этот подход скорее ближе к веб-разработке и простым приложениям.

> Масштабирование с помощью процессов

Это дополнение к 6-му пункту про параллелизм. Предлагается масштабировать приложение именно с помощью процессов. В следующих уроках мы разберём это подробнее, но в целом это про то, что нужно делать несколько контейнеров и в каждом контейнере запускать только один instance сервиса, а не пытаться распараллелить внутри контейнера.

> Утилизация

Мы разберём это далее, но опять же рекомендуем прочитать об этом на сайте: там всё очень хорошо описано.

> Паритет окружений

Контур, на котором вы тестируете приложения должен быть максимально похож на продуктивный (production) контур. Желательно, чтобы и контур разработки имел минимум отличий от продуктива.

Здесь речь идёт, во-первых, о конфигурациях, данных, и это очевидно, а во-вторых, о сторонних службах. Рассмотрим простейший пример с базами. Есть PostgreSQL-полноценная база данных, и есть SQLite-файловая база данных. Конечно, со второй работать проще, потому что она создаётся почти как файл, да и поддерживать её не нужно — фактически вы работаете с файлом. Из-за этого удобства вам может захотеться срезать угол и использовать SQLite в тестовом контуре, несмотря на то что в продуктиве — Postgres, ведь это обе SQL базы данных и синтаксис похож. В реальности все системы имеют много скрытых отличий, поэтому вы наверняка когда-нибудь пропустите в production функционал, отработавший на тестовом SQLite, но не работающий на Postgres.

Мораль проста: тестовый контур делаем таким же, как и prod (разве что не отмасштабированным), а для локальной разработки пишем скрипты, которые поднимут службы — благо сейчас поднять Postgres или Redis локально очень просто, скрипты запусков очень удобно хранятся в IDE.

> Логи пишутся в stdout/stderr

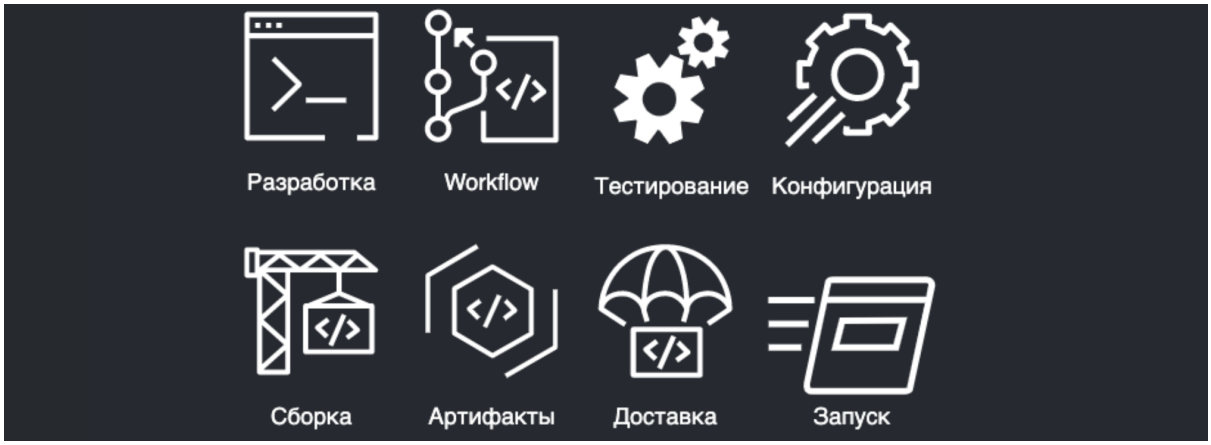
Мы также рассмотрим это позже. Вкратце: само приложение не должно заниматься маршрутизацией и складыванием логов — оно просто должно писать в свой stdout/stderr и не более.

> Процессы администрирования

Этот пункт не совсем релевантный, поэтому рекомендуем вам самостоятельно прочесть о нём на сайте.

Таким образом, мы познакомились с тем, как сейчас выглядят типичные бэкенды. Они состоят из микросервисов, которые общаются по REST API и при необходимости обладают своими базами. Также мы ознакомились с рекомендованными практиками проектирования микросервисов. Легко заметить лейтмотив в этих рекомендациях: они даются из расчёта, что приложения нужно будет переносить между окружениями, фактор масштабирования могут в любой момент как повышать, так и понижать, а упавший процесс приложения — это штатная ситуация. И это не просто так, потому что в суровом продакшене все 3 кейса достаточно часты.

> Процессы



Теперь предлагаем вам ознакомиться с общим видением процессов.

Всё, что происходит до тестирования, это разработка. Если тестирование не прошло, всё возвращается в разработку, где код должен версионироваться, а работа в системе контроля версий по-хорошему должна иметь стандарты, т.е. workflow, которые включают в себя:

- Правила работы в ветках;
- Шаблон коммитов;
- Релизную политику.

На основе этого workflow непосредственно будет настроено обновление ваших контуров. Несмотря на эту прямую связь, как мы уже выяснили, конфигурации контуров должны храниться отдельно от кода. Конфигурации контура должны объединяться с кодом контура, собираться для получения, например, докер образа и доставляться до того места, в котором workflow будет запускаться.

Какие-то вещи вам могут быть незнакомы и непонятны — это не страшно, потому что сейчас нам нужно понять, как будут выглядеть наши процессы, какие будут трудности, на что обращать внимание и как мы в целом будем мыслить, настраивая наши процессы.

> Workflow

Workflow в системах контроля версий т.е. в Git — это правила, по которым вы работаете в приватных ветках и интегрируете свои изменения в публичные ветки и согласно которым релиз выкатывается на контур. Команда, работающая над проектом, сама выбирает удобный workflow. Иногда в разных репозиториях одного проекта workflow могут отличаться, потому что над одним сервисом могут работать 5 человек и этот сервис требуется проверять во всех контурах, а

над другим сервисом работают 1-2 человека и ему достаточно находиться только в 2-х контурах — TEST и PROD.

Вот тезисно некоторые условия, которые должны обеспечиваться вашим workflow:

- Ваши feature-ветки должны создаваться от максимально актуального состояния кода, но при этом в этом состоянии не должно быть непротестированного кода;
- Каждая фича (feature) должна проходить через все регламентированные публичные ветки и не должна блокироваться другой фичей;
- Слияние публичных веток также не должно блокироваться feature-ветками;
- В продуктивной (prod) ветке должна быть такая история, которая позволит быстро откатиться на последнюю стабильную версию прода в случае появления проблем с последним релизом;
- Нужно иметь возможность выкатывать фичи пачками, т.е. собирать релиз из протестированных фич и выкатывать его на прод в удобное время.

Давайте рассмотрим пример одного довольно универсального workflow, после которого суть этих условий и способы их обеспечения станут ясны.

Пойдём с конца: вам, очевидно, нужна продуктивная ветка, из которой изменения будут катиться в продакш. Так и будем её называть: PROD. Иконка с парашютом означает, что в этой ветке настроена раскатка контура, т.е. при коммите (commit) в ветку, приложение в соответствующем контуре обновляется. Перед продом по-хорошему нужно место, где все новые фичи соберутся в релиз, ведь, скорее всего, фичи зависят друг от друга и их нельзя просто брать и по одной закидывать в продакшн. Но даже если фичи независимы, есть 2 аргумента в пользу наличия такого места перед продакшеном:

- Процесс развёртывания (выкатывания) в продакшн требует внимательного сопровождения — вы должны убедиться, что всё готово и работает, и пристально наблюдать за процессом выкатки и самим приложением сразу после выкатки, что требует много ресурсов. С помощью дополнительной ветки STAGE перед продом количество таких мероприятий сокращается.
- Если в проде выявилась какая-то проблема, и вам нужно откатить PROD до стабильной версии, то выбрать эту стабильную версию будет сложно, если в неделю вы выпускали в PROD по 6 фичей по отдельности. Если же

релизить фичи сразу все вместе раз в неделю или две, то найти последнюю стабильную версию будет значительно проще.

Итак, надобность ветки STAGE мы оправдали, Кстати, на STAGE ветке также настроено развёртывание (deploy), т.е. фактически это наш второй контур, который всегда активен и который вы всегда можете дёрнуть.

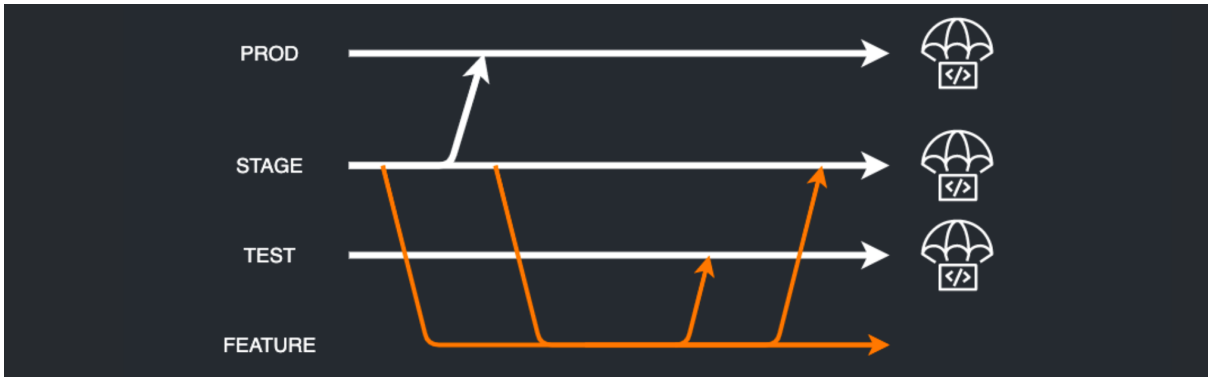
Текущая версия workflow

Конечно, ещё нужно поговорить о ветке TEST. На ней настроен всегда активный третий контур и к ней мы вернёмся чуть позже.

Допустим, разработчик приходит разрабатывать новую функцию (feature). С одной стороны, ему необходимо создать ветку от самой актуальной версии кода, чтобы не получилось так, что он разработал свою фичу, ориентируясь на устаревший функционал, а с другой стороны — нет смысла делать новую ветку (branch) от самой актуальной, но нестабильной версии, поэтому в данном случае мы ищем баланс.

В проде лежит код, устаревший как минимум на время, прошедшее с прошлого релиза, что довольно много, поэтому мы будем выбирать между STAGE и PROD. TEST более актуальный, чем STAGE, но он пока не протестированный, и начинать создавать свою фичу от него неразумно, поэтому будем создавать фичу от STAGE. Пока фича разрабатывается, в STAGE приходят новые версии кода от других разработчиков, и вы время от времени вытягиваете (pull) изменения со STAGE в свою ветку. По завершению фичи, вы отправляете её на тестирование на TEST контур, т.е. делаете merge request в ветку TEST.

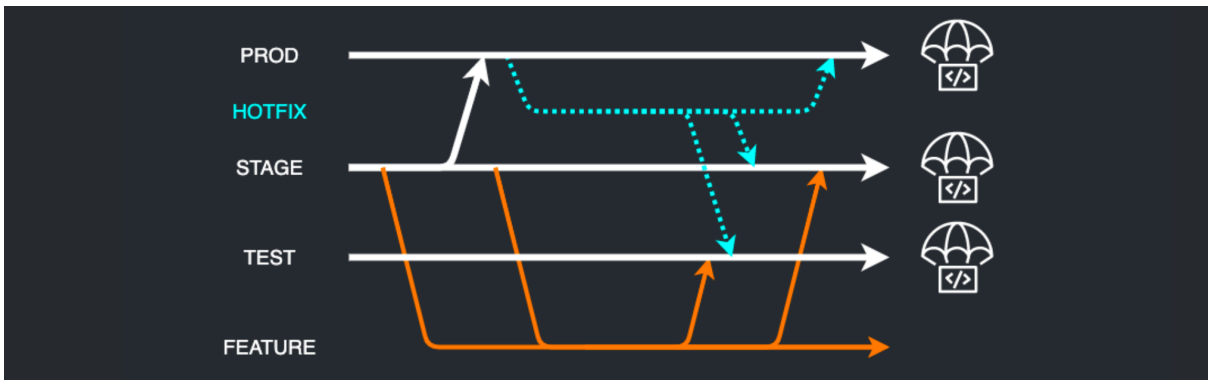
Как только новый функционал (features) протестировался (вручную или автоматически), проект мёржится в STAGE. Почему нельзя было TEST целиком закинуть в STAGE ровно таким же образом, каким мы STAGE закинули в PROD? Дело в том, что в TEST фичи могут тестироваться с разной скоростью и весьма вероятно, что какие-то из фичей в TEST содержат баги и их потом отправят на доработку, поэтому выкатывание TEST в STAGE напрямую означает выкатывание потенциально забагованных и непротестированных фич.



Обновлённая версия workflow

С основной частью workflow мы разобрались. Осталось только понять, что делать, если выявлена проблема. Откатывать production нельзя, а делать исправления, соблюдая весь порядок, нет времени.

В таких случаях делают hotfix: создают новую ветку от PROD, вносят исправления и закидывают обратно в PROD. Но ни в коем случае нельзя забывать, что эти изменения нужно закинуть в STAGE и TEST!



Финальная версия workflow

Мы закончили с workflow, который удовлетворяет условиям, сформулированным нами вначале. Если вы не можете определиться, с каким workflow разрабатывать проект, возьмите только что разобранный. Если со временем вы поймёте, что он не подходит, то ничто не помешает со следующего релиза начать работать в соответствии с другим workflow.

Перед тем как двинуться дальше, хочется напомнить, что несоблюдение workflow чревато не только усложнением процесса эксплуатации и ревизии Git, но и проблемами с CI/CD, ведь workflow лежит в его основе.

> Тестирование

Жить без автотестов очень сложно. Например, в компании Яндекс вы не сможете вмёржить вашу фичу в публичную ветку, если не напишите тесты на эту фичу. Предполагая, что ценность и польза тестов вам ясна, пробежимся по тому, что нужно тестировать.

В интернете можно найти много разных классификаций тестов, но все они сводятся к списку постоянной длины:

1. Unit-тесты
2. Smoke-тесты
3. Integration-тесты
4. Load-тесты
5. Нефункциональные тесты
6. Fuzzy-тесты
7. Регрессивные тесты
8. Прочие тесты

Скорее всего, вы уже плюс-минус представляете, что есть что. Но давайте всё равно пробежимся по применимости и важности отдельных видов тестов.

> Unit-тесты

В первую очередь стоит писать основные юнит-тесты (Block и пр.). Они пишутся быстро и сразу дают понимание, что именно сломалось. Юнит-тесты просто автоматизировать, поэтому можно не жадничать и втыкать их везде, просто гоняя их локально, что существенно ускоряет процесс разработки. Понятно, что абсолютно всё покрывать юнит-тестами не стоит — необходимый процент покрытия определяется на глаз.

> Smoke-тесты

Их также называют healthcheck-тестами. Точного определения нет, но это элементарные проверки/тесты, которые могут показать, что софт не совсем хорошо работает. Например, если тестируется веб-сервис на TECT контуре, то после релиза можно в пайплайне дёргать самые простые эндпоинты и проверять, что они отработали и "ошибок нет". Или если это не веб-сервис, то, например, можно проверять, что количество выгруженных предсказаний совпадает с количеством клиентов во входных данных. Таким образом, мы

отлавливаем критические проблемы, проявляющиеся на старте приложения и связанные чаще всего с развёрткой и/или конфигурированием, а также всё, что могли пропустить unit-тесты. На самом деле smoke-тесты можно делать даже перед unit-тестами.

> Integration- / UAT-тесты

Далее идут Integration / UAT (User Acceptance Tests) тесты. Интеграционным тестами называют как совместное тестирование модулей внутри приложения, так и совместное тестирование приложений. Такие тесты индивидуальны, и писать их сложнее, чем юнит-тесты, как минимум потому, что они часто запускаются не только на уровне кода приложения, но и на уровне инфраструктуры. Практика показывает, что, например, вы можете иметь достаточное тестирование компонент по отдельности, что позволит прожить без интеграционных тестов очень долго.

В микросервисной архитектуре интеграционные тесты превращаются скорее в контракт-тесты (контракт — это соглашение формата данных, передающихся через REST API). Писать тесты на контракты в принципе уже проще и понятнее, к тому же есть специальные инструменты для контракт-тестов (Pactflow). А вот в UAT-стенде (имеется в виду стенд для приёмочного тестирования) потребность в UAT-тестах может возникнуть с первых дней (и это особенно справедливо для ML сервисов). UAT-стендом может быть ещё один стенд между TEST-стендом и STAGE-стендом — и тогда UAT-стенд будет полностью дублировать TEST-стенд.

> Load-тесты

Load тесты писать достаточно просто. Они служат для того, чтобы проверять, как хорошо ваш сервис справляется с большой нагрузкой. Такие тесты, как правило, не идентифицируют проблему.

> Fuzzy-тесты

Интересный вид тестирования — fuzzy testing, в котором мы как бы пытаемся сломать приложение. Например, можно попытаться произвести SQL инъекцию или случайным образом удалить строки из базы, чтобы протестировать работу вашего приложения в нештатных случаях. Netflix выпустил инструмент Chaos Monkey, который случайным образом гасит виртуалки в вашей инфраструктуре, побуждая вас разрабатывать отказоустойчивые сервисы.

Очевидно, что ни в Fuzzy, ни в каких-либо других тестах не следует переходить границу и пытаться тестировать условные Postgres или Redis — для них есть свои тесты.

> Нефункциональные тесты

В нефункциональные тесты входят тесты на нагрузку и скорость ответа. Они понадобятся не сразу, и, вообще говоря, их можно не автоматизировать и запускать только по необходимости (в репозитории просто будут лежать скрипты с тестами).

> Регрессионные тесты

В регрессионных тестах вы проверяете, не сломался ли после ваших изменений старый функционал. То есть если на этот старый функционал тоже были написаны тесты, которые запускаются разом со всеми остальными, то можно сказать, что вместе, допустим, с unit-тестами вы заодно проводите и регрессионное тестирование.

> Прочие тесты

Остальные виды тестирования встречаются гораздо реже. Те, что были перечислены выше, есть в том числе и в нашем продукте.

> Continuous Integration / Continuous Deployment (CI/CD)

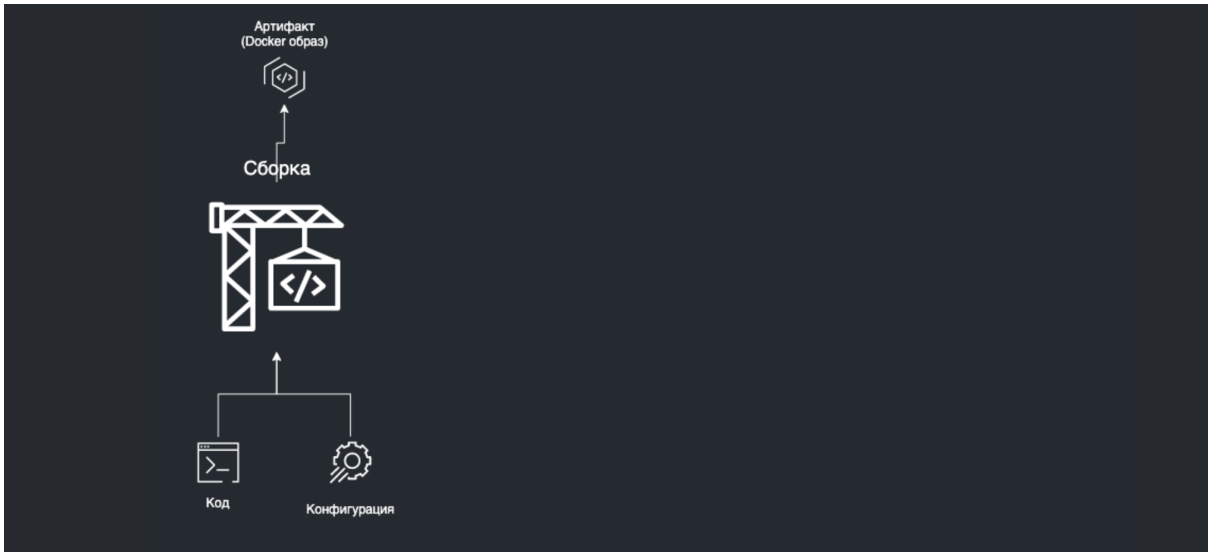
Если вы ещё не знаете, что это такое, то лучше поскорее узнать :) Мы довольно подробно познакомимся с GitLab CI/CD на занятиях, но к этим занятиям желательно уже понимать, о чём идет речь. Для этого можно посмотреть какое-нибудь введение по GitLab CI/CD на Youtube.

Здесь всё будет достаточно кратко. В вашем репозитории вы описываете файл с инструкциями, которые будут запускаться при каждом вашем коммите (commit). Инструкции могут быть абсолютно любые, но как правило, это инструкции по сборке/тесту и т.д. Каждый ваш коммит отслеживается CI системой, которая читает ваш файл с инструкциями. Затем система берёт ваш код, доставляет его на специально выделенную вычислительную единицу и на этой единице выполняет с кодом описанные вами инструкции.

Речь идёт о следующих инструкциях:

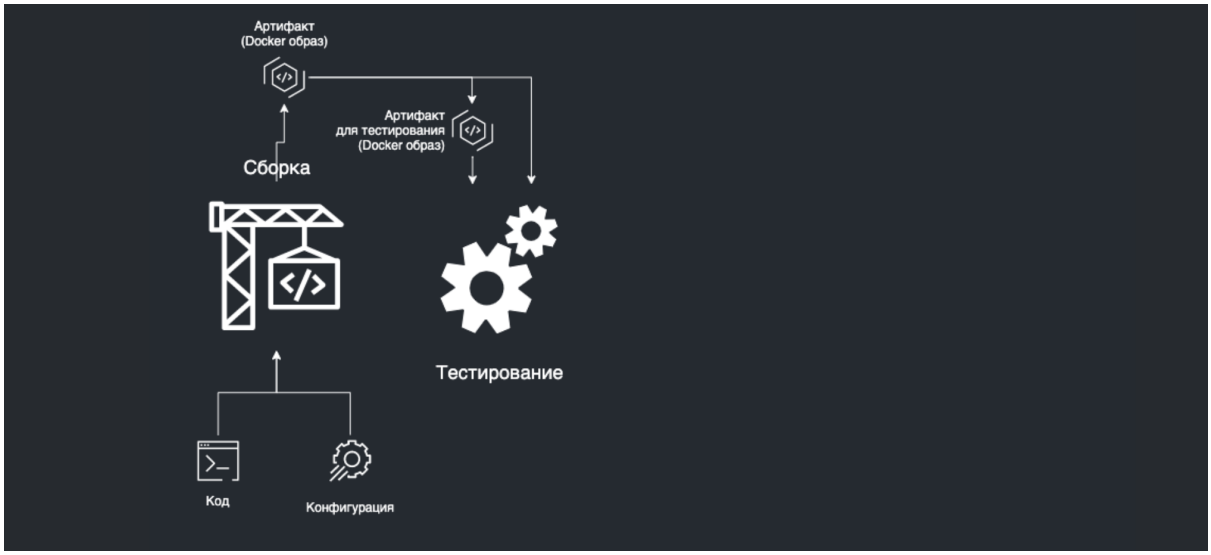
- Сборке;
- Тестировании (юнит-тесты, smoke-тесты, integration-тесты);
- Доставке кода и сборок;
- Запуске кода и сборок;
- Другие специфичные вещи.

Все эти процессы просто автоматизируются в CI/CD и, конечно, могут существовать в команде в неавтоматизированном виде. Очевидно, автоматизация — это удобно. Рассмотрим каждый этап по отдельности.



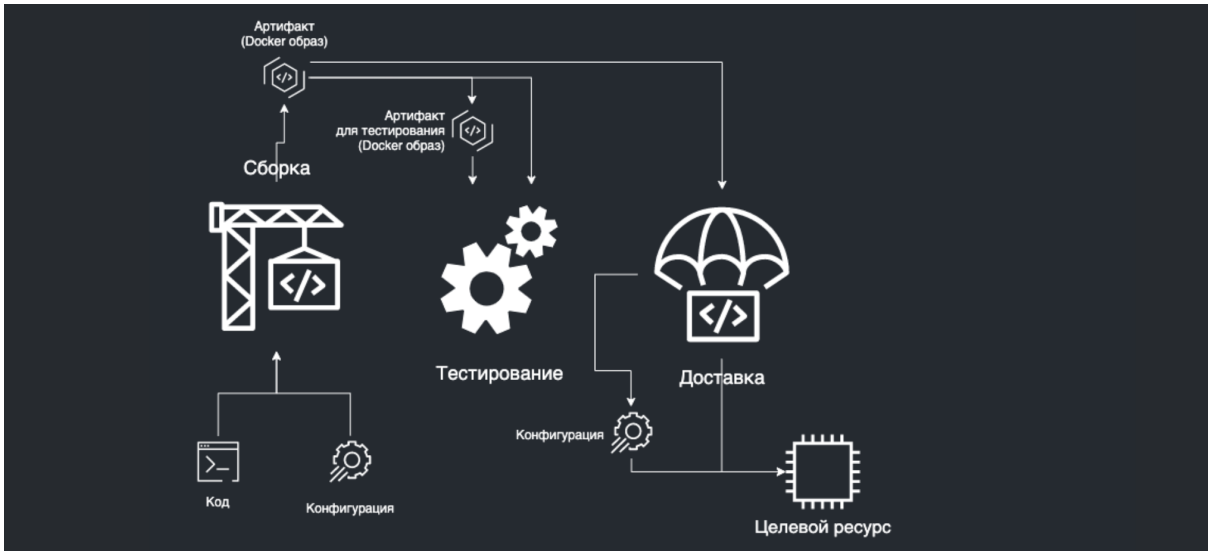
Этап сборки

На этапе сборки вы, скорее всего, собираете Docker-образ и загружаете его в registry образов. Registry образов — это хранилище образов докера. На схеме оно не изображено, но считайте, что Docker-образ находится именно в registry. Также можно собирать не Docker-образ, а pypi пакет в качестве альтернативы. В некоторых компаниях собирают даже rpm пакеты (которые устанавливаются через пакетный менеджер операционной системы) или просто формируют zip-архив. Конечно, чаще всего используют Docker-образы, но zip-архивы, pypi пакеты и артефакты из других языков (например jar файлы в java) тоже используются. Также в каком-нибудь жёстком enterprise используются rpm-пакеты, но это бывает довольно редко. Уже на этом этапе в сборку можно включить config-файлы, пароли, ключи, что будет правильным решением в одних случаях и неправильным — в других. Мы вернёмся к этому вопросу в рамках нашей практики.



Тест сборки

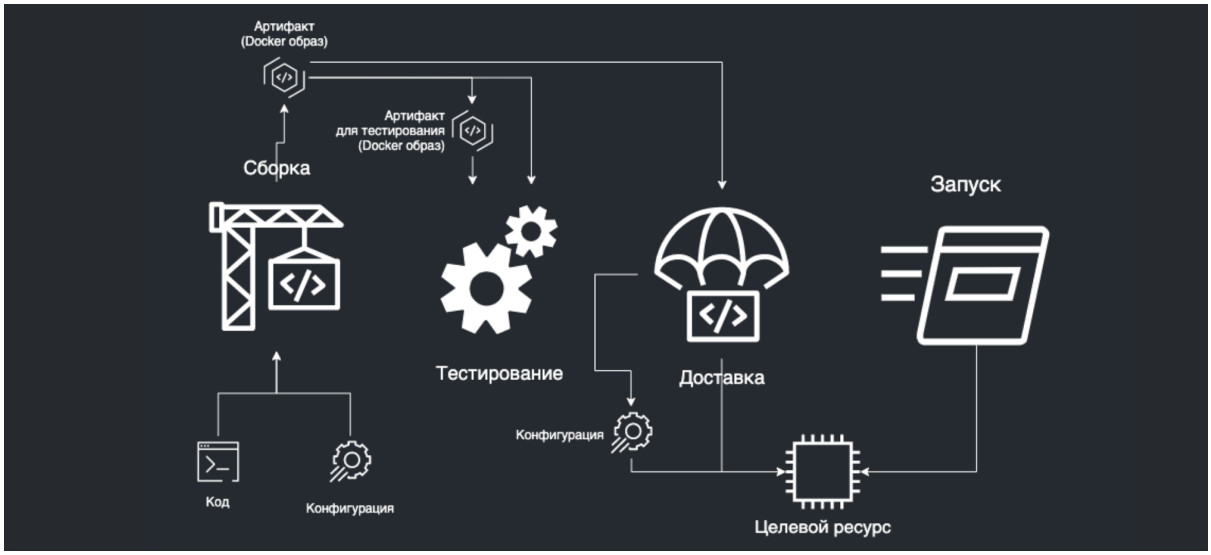
После того как сборка готова, вы запускаете тесты этой сборки. Например, если в собранном вами Docker-образе в качестве entrypoint была команда запуска веб-сервиса, то вы можете запустить тесты из этого же образа, переопределив entrypoint с запуска веб-сервиса на запуск тестов. Таким образом, собрав всего один образ, вы сможете и прогнать тесты, и использовать образ для запуска в дальнейшем. Но для этого нужно, чтобы тестирующий фреймворк был установлен в образ. В этом ничего плохого нет, но существует один постулат, который звучит следующим образом: "Не храните в сборках ненужные для runtime зависимости". Те, кто максимально строго следуют этому постулату, могут сразу после сборки основного образа собрать на его основе второй образ специально для тестирования, доустановив туда pytest и переопределив entrypoint (т.е. запускающую команду), тем самым освободив целевой образ от ненужной зависимости. В принципе, тесты можно запускать и до сборки, тогда, если тесты не прошли, можно будет остановить пайплайн и не тратить время на сборку. Но для этого нужно, чтобы раннеры (т.е. вычислительные единицы, на которых выполняются команды) были с необходимыми зависимостями под каждый проект, что сложно обеспечить из-за ресурсоемкости. Также вы можете установить эти зависимости самостоятельно, но это займёт ещё больше времени. В то же время с тестированием сборки вы обойдётесь одним раннером с докером на все проекты, поэтому чаще выбирается именно вариант с тестированием сборки.



Этап доставки

На этапе доставки вы вероятно делаете что-то одно из трёх:

- По SSH ходите на целевые машины и выполняете удалённые команды, например pull докер-образов из registry. Таким образом, в целевом окружении, появятся сборки, которые потом можно запустить;
- Запускаете ansible playbook, в котором прописаны все инструкции по доставке кода/сборки;
- Используете Kubernetes API — kubectl, сразу делая deploy ваших контейнеров в кластер Kubernetes. В этом случае процесс доставки и процесс запуска приложения неотделимы и выполняются за один раз. На схеме видим, что на этом шаге может выполняться доставка файлов конфигурации в целевое окружение. Вероятно, эту доставку мы будем делать, если решили не вшивать файлы конфигурации в сборку.



Этап запуска

С точки зрения CI/CD запуск (выкатка) похож на доставку — на этих фазах используется приблизительно тот же набор подходов и инструментов. Всё точно так же, как и в доставке: по SSH или через агента мы ходим в целевую среду исполнения, чтобы запустить сборки, которые уже там находятся благодаря предыдущему шагу.

В Git всегда остаются коммиты, на которых запускались CI/CD job'ы (т.е. пайплайны), и вы в любой момент времени можете вернуться к старым пайплайнам и перезапустить их, поэтому они проектируются идемпотентно. Это означает, что пайплайн, будучи запущенным второй раз, не внесёт каких-либо дополнительных изменений в систему и выполнится точно так же, как и в первом запуске. Т.е. не стоит делать пайплайны, job'ы которых зависят, например, от данных в базе или других репозиториев. Конечно, вы можете использовать сабмодули (что крайне не рекомендуется) или собирать в пайплайне пакеты из исходных файлов (sourse'ов), но тогда необходимо иметь возможность протестировать их в пайплайне в тех же unit-тестах или integration-тестах. Иначе у вас на карте будут тёмные места, проблемы в которых не будут замечены вовремя.

> Конфигурация

Первое, с чего все начинают, — конфиги (файлы конфигураций) в репозитории в файле `conf.txt`. Это проверенная технология, но получается, что вместе с кодом версионируются и конфиги. Это может привести к merge-реквестам, в которых придётся разрешать конфликты конфигураций, а если у вас настроен

CI/CD, то каждое изменение конфигов будет сопровождаться запуском пайплайнов.

Когда пайплайны настроены правильно, ничего критичного в этом случае не произойдёт, если у вас хватает железок и времени гонять пайплайны лишний раз, ведь пайплайны могут быть долгими, а сборки — тяжёлыми. Также может показаться удобным наличие готовой сборки на каждый набор конфигов. То есть фактически тут 4 преимущества:

- Это простой подход;
- Удобно держать файл конфигурации около кода (причём в каждом контуре он свой);
- Происходит версионизация конфига;
- При обновлении конфигурации автоматически раскатывается новая версия.

Последний вариант, как вы понимаете, не удовлетворяет принципам 12 factor apps. Посмотрим в сторону других вариантов.

Один из них — хранение файлов конфигураций прямо на месте развёртывания. В простейшем случае конфиги исправляются руками в папке, в которую код доставляется и в которой он запускается. Если всё запускается в контейнерах, то нужно монтировать файл с конфигами как bind mount или как volume. Также есть оптимальный по соотношению удобства и приложенных усилий вариант — держать значения конфигов в CI/CD переменных и рендерить конфиг файл по шаблону в CI/CD пайплайне, например на этапе деплоя, а затем либо использовать этот конфиг в команде запуска, либо доставлять его как файл и опять же монтировать к контейнерам.

Этот подход обладает следующими преимуществами:

- Нет дополнительной нагрузки на workflow;
- Нет запусков пайплайнов на каждое изменение конфигурации (не стоит недооценивать этот пункт);
- Выше уровень безопасности (все, кто имеет доступ к Git, смогут увидеть чувствительную информацию в первом способе);
- Более быстрый перенос приложения и замена ресурсов.

В данном случае недостатки одного подхода являются преимуществами другого.

Если оптимального варианта уже недостаточно или вы педант, то можно перейти на системы контроля конфигураций. В модуле рассматривать мы их не будем, но для педантов оставим ссылку с кратким обзором инструментов. Скажем только, что наиболее распространёнными являются Ansible, CHEF и SALTSTACK. В статье ничего не говорится про Consul, но это уже больше чем простой менеджер конфигураций. Кстати, есть ещё "дедовский" способ с хранением всех конфигураций в отдельном репозитории, но те, кто его когда-либо использовали, обычно отказывались от этого способа.

> Развёртка

Развёртывание софта — это процесс его доставки и запуска на целевых ресурсах (т.е. серверах, облаках и даже контейнерах). Мы с высоты птичьего полёта осмотрели этот процесс с точки зрения CI/CD пайплайна. Но CI/CD — это просто автоматизация наших процессов. Сейчас же чуть-чуть покопаемся в сути процесса развёртки.

В первую очередь необходимо определить, должны ли наши приложения разворачиваться без простоя (zero downtime deploy), т.к. это накладывает определённые ограничения на разрабатываемый нами софт (например, вышеупомянутые 12 факторов) и делает процесс деплоя сложнее, или всё-таки простой нам не страшен. Если даже простой не страшен, то в любом случае придётся минимизировать время простоя, а работы проводить в запланированные технологические окна таким образом, чтобы клиенты этого не замечали и не страдали. К тому же не всегда возможно делать zero time deploy — например, могут помешать блокирующие миграции баз данных.

Давайте рассмотрим самый простой вариант развёртки с учётом того, что доставка сборок уже совершена. Он будет состоять из следующих шагов:

1. Остановка предыдущей версии приложения;
2. Миграции данных, обновление БД и прочих служебных операций (по необходимости);
3. Запуск новой версии;
4. Проверка, что всё прошло корректно.

Достаточно тривиальный алгоритм, который легко реализовать на любом скриптовом языке, например Bash или Python. Также можно взять автоматизации вроде Ansible.

С бесшовным деплоем уже будет поинтереснее. Существуют несколько стратегий для развёртки с нулевым простоем:

1. Green/Blue или Red/Black;
2. Rolling;
3. Канареечный (Canary).

При использовании этих стратегий уже нет необходимости производить полную остановку приложения. Поговорим о том, как они реализуются.

Самая простая стратегия — Green/Blue. Её суть в том, что мы поднимаем полную копию окружения новой версии, пока всё ещё работает старая. Как только мы понимаем, что новая копия готова к приёму трафика от наших клиентов, мы переключаем трафик на новое окружение, а старое выключаем. Здесь в качестве переключателя, как правило, используется балансировщик трафика или proxy-сервер вроде nginx, в бэкендах (или upstream'ax) которого указаны наши оба окружения.

Второй вариант — это Rolling. Это общая концепция: если у нас есть несколько реплик одного приложения, то мы по одной каждую реплику заменяем на новую версию.

Как частный случай Rolling можно ещё выделить канареечный (Canary) релиз. Идея в том, что мы точно так же делим весь процесс обновления на маленькие шаги. И на каждом шаге увеличиваем количество трафика, которое направляется на обновлённую версию окружения. При этом мы проверяем, что у нас всё идёт штатно и ничего не ломается. Если же в какой-то момент времени что-то идёт не так, мы получаем сигнал от системы мониторинга и откатываем обновление. Конечно, самыми безопасными способами являются A/B и канареечный релизы.

Итак, теперь мы разобрались и с процессами: если приложение спроектировано правильно, то его можно развёртывать, используя разные стратегии, которые автоматизированы в CI/CD пайплайне. В этом же пайплайне мы собираем, тестируем и доставляем приложения, а в репозиториях живём по workflow.

Из базовых вещей в бэкенде мы не пробежались разве что по миграциям для баз данных, которые для ML бэкендов имеют малую релевантность. Поэтому теперь, имея все эти знания, мы можем порассуждать, что из рассмотренного пригодится в ML бэкендах, а чего пока не хватает.

> Резюме и дополнительные материалы

Таким образом, поскольку к продакшену пайплайновых систем требования не такие высокие, как к продакшену микросервисных, мы будем больше говорить о вторых. Подробно поговорим о коде приложения, правильной контейнеризации и процессах в CI/CD.

В пайплайновых системах обычно всё поддаётся автоматизации в Airflow, что очевидно проще, чем разработать устойчивую realtime систему. Кроме того, и у первых, и у вторых систем есть общая часть с обучением и дообучением моделей, а также их хранением, которой мы не коснулись, но это офлайн часть жизненного цикла, имеющая мало общего с продом, и мы до неё ещё дойдём.

- Про микросервисы
- Видео про построение систем и Cap Theorem
- Видео про архитектуры (довольно водянистое)