Лекция 14: Continuous Delivery

Это подход, направленные на решение проблемы ознакомления с последней версией. Допустим, если вы в данный момент работаете над новой версией, но при этом нигде не сохранена старая. Или же Вы выпустили новую версию с грубой ошибкой из-за которой, Вы теряете клиентов, и Вам срочно нужно откатиться назад, а сборка занимает много времени.

Выпуск новых версий

- Как быстро изменения в 1 строчку кода попадают к пользователям?
- Сколько для этого требуется усилий?
- Насколько это процесс повторяем?
- Сколько человек вовлечено в этот процесс?

Итак, значение ответов эти вопросы очень сильно влияют на наличие клиентов.
Раньше релизы делали примерно раз в год, и поэтому было время чтобы к ним тщательно подготовиться и все знали, когда можно будет посмотреть новую версию. На данный момент многие сервисы популярные делают несколько релизов в день. Очевидно, что чтобы так делать нужна отлаженная автоматизированная система выкатывания релиза.
Основная мысль этой лекции заключается в том, что если вы уже настроили процесс разработки то на этом все не заканчивается, вам нужно уметь этот код еще и доставлять пользователям. Мы сегодня рассмотрим организационные и технические методы, которые помогут с этим.

Антипаттерн управления релизами №1: ручное развёртывание

- Необходимость подробного документирования процесса
    - или эксперта
- Повышение стоимости и времени развёртывания
    - дополнительная нагрузка на специалистов
- Непредсказуемость процесса
    - отличия в конфигурации приложений или окружений
- Отсутствие гарантий надёжности и повторяемости

Итак, как делать не надо! Во многих компаниях до сих пор процесс развертывания вручную, это значит что в случае мобильного приложения - это будет выкладывание на AppStore. А это значит, что нужно отдавать работу выкладывания отдельному человеку, в принципе ничего плохого нет, но повышается Bus Factor, потому что только этот человек знает где лежат сертификаты.
Однако, представим что у нас не один файл, а целая распределенная система, в которой надо выложить не только бинарники, но и в определенной конфигурации, запущенными

под определенной операционной системой с установленными всеми необходимыми библиотеками. И это все руками делать - задача не для слабонервных.
Потому что нужно найти Primary релиз, упаковать его. Дальше настроить окружающую среду, установить все библиотеки, распаковать его в нужные места, поднять все БД, настроить все настройки, настроить все интерфейсы.
Иногда для этого заводят целый отдел, это как бы системные администраторы которые раскатывают ваше приложение по серверам. Но опять же больше никто делать это не может. И чаще всего у них есть подробная документация развертывания, и это становится достаточно рутинная работа, однако требующая высоких специалистов.
Также понятно, что все ручные действия завязаны на человекаа, а человек может ошибиться, опечататься и т.д.
Таким образом этот процесс становится не часто повторяемым и непредсказуем.

Антипаттерн управления релизами №2: развёртывание только после завершения разработки

- Все ошибки проявляются впервые
    - включая неверные предположения о системе и её окружении
- Специальная команда из разных специалистов
    - между которыми нужно наладить взаимодействие
    - “оторванность” разработчиков от реальной жизни
- Попытки срочных “горячих изменений”

Вторая ошибка, которую можно совершить - это развёртывание только после завершения разработки. В данном случае имеется в виду завершение разработки - это окончание какой-нибудь версии. То есть разработчики заканчивают работу и отдают все тестировщикам которые сидят это впервые, и должны все это как-то выложить в среду близкую к релизной.
На самом деле не всегда можно выложить сразу в продакшен, но почти всегда есть стеджинг окружение, это как продакшен, но только у нас. И вот, в этом стэджинге версию рассматриваю тестировщики, они естественно могут заметить какие-нибудь ошибки, более того, может оказаться, что у разработчиков были не те представление об окружающей среде или неправильные архитектурные решения были приняты. И все эти ошибки видятся в первый раз.
И так как всегда мало времени - то это пытается все как-то костылями заменить и могут даже менять все это после релиза, Но часто эти изменения делают тестировщики и разработчики не узнают об этом. И как итог - в новой версии будут те же ошибки.
Таким образом, код который живет в репозитории и код, который работает в продакшене- получаются разными. То есть очень плохо, когда люди которые раскатывают - разные. И чтобы исправить эту ситуацию нужно пытаться делать релизы раньше, чем закончиться разработка, чтобы баги фиксили сами разработчики.

Антипаттерн управления релизами №3: ручное управление окружением в продакшене

- Ручная подготовка окружения
    - документация, эксперты, …
- Отсутствие повторяемости процесса размещения
    - сложно/невозможно откатиться к предыдущей (стабильной) версии
- Различия в окружениях разработки и продакшена
    - фиксы багов
    - изменения настроек БД, серверов и т.п.
    - переменные окружения
- Никто толком не знает, что сейчас в продакшене

И третья ошибка, которую часто допускают - в общем-то управление окружением в продакшене очень часто делается не в тот момент, когда мы разворачиваем версию, а “походу”. Меняются IP-адреса, меняются БД, меняются соединения и т.д. И изменения делаются уже на тех версиях, которые работают. Таким образом формируется ситуация когда никто толком не знает какая версия сейчас работает у заказчика. Хорошо если есть понимание версии кода, или БД, однако, версионировать окружение - очень сложно (непонятно как). И что самое страшное - у Вас может получится ситуация, когда у Вас была работающая версия, Вы ее неким способом обновили, и она перестала работать, потому что вы поменяли окружения. И Вы уже не можете откатиться. Обычно все это происходит в условиях жесткого цейтнота.
И что же с этим делать - выкатывать новые версии часто и автоматически.
Сложно недооценить эту идею: потому что автоматизация всегда очень привлекательная - скрипты всегда делают то, что они делают, никогда не ошибаются, и если Вы один раз наладили скрипт - то он будет всегда делать все правильно.
Зачем делать часто релизы -потому что Вы получаете быструю обратную связь от пользователей. О сути все Ваши артефакты можно поделить на 4 группы:

1. Бинарники
2. Настройки окружения
3. Данные
4. Конфигурации бинарников

Соответственно изменение в любом из этих артефактов - должно получать фидбэк: Если Вы поменяли какие-то конфигурационные файлы, Ваша система должна автоматически все собрать, прогнать тесты и сказать, хорошо ли все отработало. Т.е. Обратная связь должна быть как можно скорее и должна быть на любые изменения. И о наличии ошибок тоже хочется узнавать как можно сокрее. И чем больше мы тратим временем на проверку релиза - тем больше у нас к ним доверия. И мы хотим иметь механизм, который будет говорить на каком этапе произошли ошибки и отследить какие ошибки произошли.

Понятно, снижается количество ошибок и при этом люди, которые занимаются развертыванием теперь могут заниматься более полезными делами. Например тестировщики ждут когда им дадут код, ждут администраторов, которые должны дать разрешение на запуск в продакшене. Эти люди и при автоматическом релизе нужны и задействованы. Однако, они уже не делают много рутинной работы. При этом есть больше возможностей - теперь можно откатываться по версиям.
И понятно, что теперь не нужно нервничать, потому что когда ты сам разворачиваешь сервис - очень страшно допустить малейшую опечатку.

Принципы непрерывного развёртывания

- Повторяемый, надёжный процесс выпуска версий
- Максимальная автоматизация
- Максимальное версионирование всего
- Если где-то есть проблема, делаем это чаще

  Довольно распространённый принцип (типа у спортсменов). Если проблемы с тестированием -- тестируем чаще, с выкатыванием версий -- выкатываем чаще. Будет сложнее, будет больше проблем, больше стресса -- это побудит вас решать проблемы. Если процесс релиза занимает 5 дней и 10 человек, то это плохо. И вот исправлять они это будут только тогда, когда они задолбаются -- а если они будут делать это раз в месяц, то они поговорят, что “да, надо бы”, но на этом все и закончится.

  Это не самоцель -- автоматическое и частое развёртывание, -- а средство сэкономить силы и средства, потому что люди ошибаются и делают что-то неточно и невоспроизводимо.
  Пусть у вас есть проблема с раскатыванием версий. Вы знаете, как её решить, но этого никто делать не хочет -- тогда вы начинаете делать это сами, и через какое-то время все видят, что это на самом деле реально, возможно, под это выделят ресурсы. Кстати, с нуля это наводить проще, чем встраивать в живущую систему.

  Нормальное решение приходит тогда, когда кустарным методом уже просто никак не получается. [это про плохо налаженный процесс превратить в хорошо налаженный].

- Ориентация на качество
- “Сделано” значит “уже находится в релизе”

  “Definition of done”. Немного радикальная точка зрения. Не всегда это возможно, если, например, релизы только раз в неделю. А вы хотите релизиться чаще (зачем-то), то вы можете релизить не в продакшн, а где-то у себя локально.

На 57% сделано не бывает -- либо сделано, либо нет. Если сделано, то этим кто-то может пользоваться, вот оно лежит.

- Выпуск версий -- общая ответственность

  Это некий общий подход, а значит люди должны его поддерживать и как-то работать соответствующе. Это должна быть общая политика компании.

- Постоянное улучшение

Конфигурационное управление

Очень важный принцип.

- Использование систем контроля версий

  Обычно тут ставят равенство. Ну, такое: без них, конечно, нет конф управления, но не VCS едиными. Очень важно, чтобы там всё лежало и всё было обозначено.

- Управление зависимостями
  - сторонние библиотеки и внутренние зависимости

    Вопрос спорный. Кто-то говорит, что всё должно браться с централизованных серверов, потому что там все одинаковое, стабильное. Кто-то не верит серверам Maven'а и кладёт себе в Git. Это решение каждым конкретным проектом под себя принимается. В любом случае, имеет смысл делать олкальные кеширующие сервера (даже если вы пытаетесь выкачать весь интернет -- а то иначе долго каждый раз качать). Да и просто: версии могут быть разными, интернет отвалился, удалился нужный пакетик.

- Управление конфигурациями ПО
  - конфигурации и гибкость

    Степень конфигурируемости приложения. Есть такой антипаттерн, когда вы пытаетесь законфигурировать приложение в усмерть. С этим надо очень аккуратно. Ваш конфиг -- как-то меняет код вашей программы, а проверок компилятором там сильно меньше. И на содержимое конфига тестов не пишут. А если кто-то в нём опечатается, то приложение может сильно не оценить. Поэтому все надо проверять и не выносить столько много, сколько иногда хочется.

  - типы конфигураций

* Система сборки что-то делает (Maven читает конфиг и вбивает константы в код). Нехорошо так делать, конечно: вам такой бинарник не протестировать в лбьом окружении.
* Может попадать в виде бинарника
* Во время runtime'а

- Управление окружением ПО
  Есть смысл версионировать, класть в гит какие-то важные настройки. Сейчас почти все берется из образов докер-контейнера, но и его имеет смысл тогда в гит положить. Это просто прогнозируемости добавит.
  - автоматизация создания и настройки

CI

Важная практика. Чтобы она работала -- непрерывная интеграция кода -- ...

- Необходимые условия
  … должна быть система контроля версий.

  - регулярные коммиты (и мерджи)

    Если вы пользуетесь ветками, но про них отдельно.

  - вменяемая система автотестов
    Которые будут проверять систему перед приёмом

  - вменяемый по времени процесс сборки и тестирования
    Он должен быть не очень длинный. По полчаса никто не выдержит сборку ждать

  Если у вас все это есть, то вы сможете построить сбее КИ и быстренько следить, насколько нормальный код вы каждый новый раз пытаетесь закоммитить.

  КИ из книжек не совсем такой, какой обычно используют люди. Во-первых, он не дружит с ветками никак. Там только одна ветка. Мы непрерывно интегрируемся со всеми изменениями всех разработчиков. ПРоблема с ветками в чём: когда у вас есть долго живущая ветка, долго-долго назад отошедшая от мастера, и есть другие, которые периодически что-то вливают в мастер. Это ещё не страшно, вы можете эти изменения себе утаскавать. Но если вы не одна такая долгая ветка, то вам с другими такими же (и вашим собратьям с вами) будет потом очень болььно мерджиться. И идея в том, что все выкладывают в мастер. Казалось бы, это должно спровоцировать адский хаос: мы же никогда не получим нормальный результат, код же потеряет целостность -- но считается, что изменения должны быть завершёнными, не должны ломать билд, они работают и проходят тесты. И

это должно научить разработчиков вот так вот по чуть-чуть работать всем вместе.

Если честно, то в студенческих проектах это не особо просто внедрить. Не всегда получается. Схема с ветками живёт лучше, даже несмотря на конфликты при мердже.

Итак, у нас есть билд, который не ломается -- посмотрим на полезные практики.

- Полезные практики
  - не коммитим в сломанный билд

    Человек срочно-срочно что-то делает и чинит, то никто больше не коммитит -- даём человеку спокойно закончить. Но это влечёт и то, что есть ограничеине на время починки. Если слишком долго то ты откатываешься.

  - проверяем все тесты локально перед коммитом

    Проблема в том, что локально тесты могут выполняться довольно долго. И модульные тесты ещё ничего, а вот взаимодействие файлов между собой, проверка графического интерфейса -- их вряд ли разумно проверять перед уаждым коммитом.

    - проверяем мердж мастера в локальную ветку (если работаем с веткой)

      Если вы все-таки используете ветки, то есть смысл регулярно забирать изменения из мастера себе. И лучше чаще раза в день -- на каждое обновление.

      Не всегда удаётся избежать веток просто потому, что некоторые изменения неизбежно ломают код во многих местах. Например, переход на новую версию библиотеки с новыми калссными фишками. Вы поломаете всё, неизбежно, но только один раз.

  - проверяем билд на сервере перед новой задачей

    До тех пор, пока билд не прошёл, за новую задачу браться не рекомендуется. Тестирование может занимать много времени. Если оно занимает три часа, то окей, вы не будете ждать и пойдёте дальше. А если минут десять, то проще подождать, чем переключать контексты

ментальные.

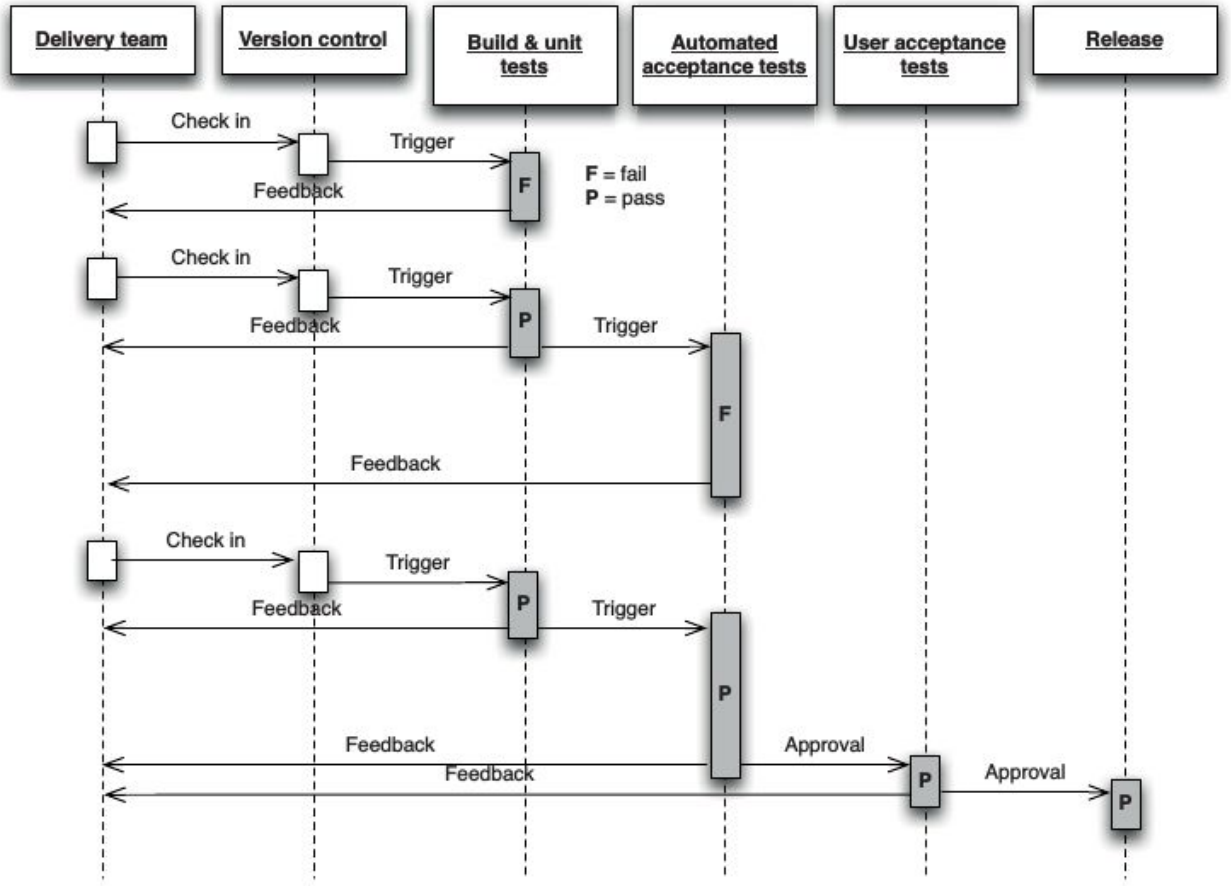
- сломанному билду нет оправданий
  - кто сломал, тот и виноват

    Винить всех вокруг -- подход неправильных программистов. Твои изменения что-то сломали, так что это ты должен это что-то чинить. Билды ломаются -- это нормально. Но билды не должны ломаться надолго; вот оставлять их на выходные -- ненормально. ПОтому что пока ты это не починишь,никто продолжать не может. Ты просто не должен мешать другим работать своим сломанным билдом.

  - всегда будь готов откатить изменения
  - временной лимит на починку билда
- настраиваем систему оповещений

  Не обязательно именно электронная -- хоть колокольчик, хоть плакат вешать. Но о проблеме должны узнать сразу все, инчае общий процесс работы нарушится. Но обычно, конечно, настраивают систему электронных сообщений. Ну и почти всегда есть dashboard, где показано, кто где работает и что где сломалось.

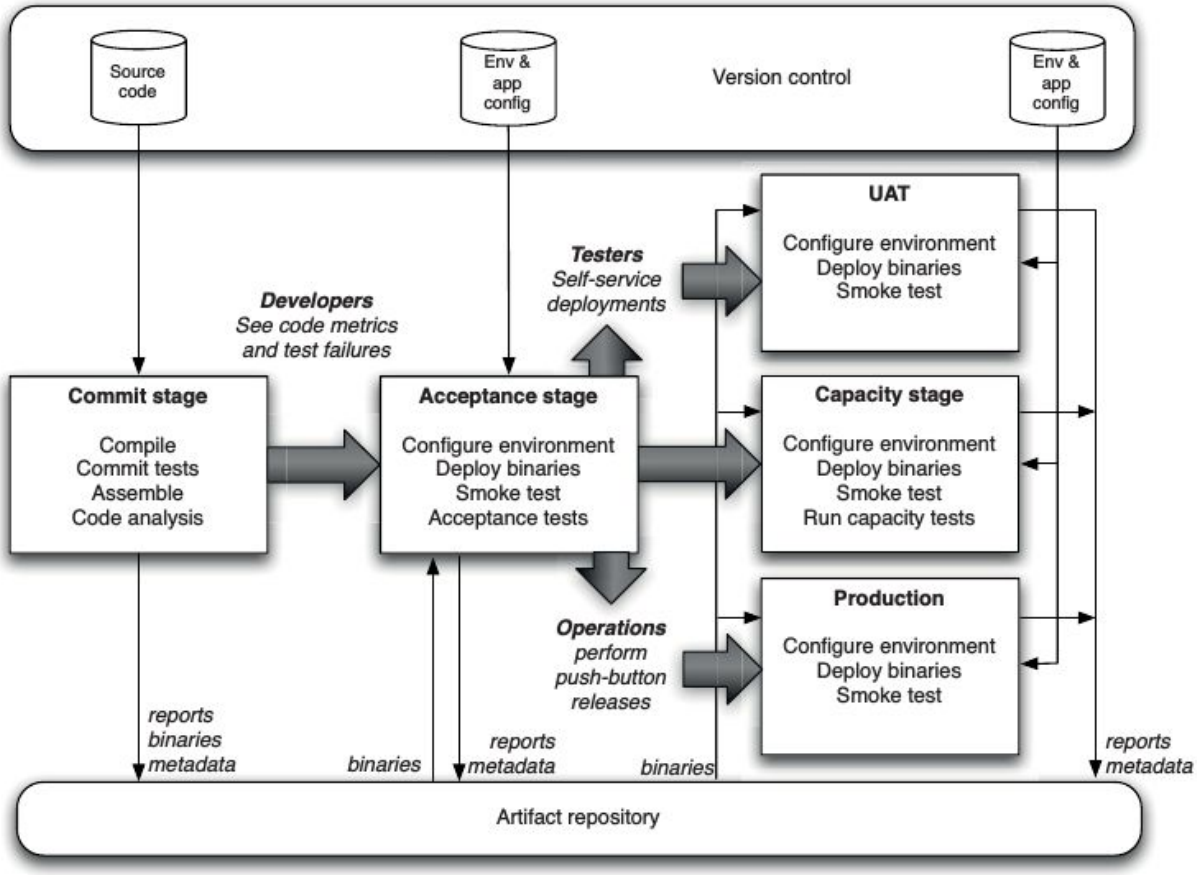
[Я ЗАКОНЧИЛЬ]



По хорошему, весь конвейер состоит из этапов:

- Изменение в системе контроля версий.
- Запуск сборки и модульные тесты.
- Автоматические еще более сложные тесты.
- Далее это все может попадать к тестировщикам для ручного тестирования.
- И наконец оно должно попадать к администраторам, для развертывания в продакшен или куда-то еще.

Задача автоматизации состоит в том, чтобы каждый следующий этап запускался автоматизировано или почти автоматизировано по окончанию предыдущего этапа. Понятно, что не все стадии должны запускаться автоматически. Первые две явно должны, а вот последние две уже явно нет. Но количество усилий, которое должны затратить квалифицированные специалисты на те или иные действия должны быть сведены к минимуму. В идеале, это когда есть доска, на которой видны состояния всех версий, а дальше все делается автоматически, по нажатию нескольких кнопок.



Более сложная версия pipeline выглядит примерно так, как показано на рисунке. Естественно, для каждого проекта и для каждой компании этот конвейер будет совершенно различаться. Просто потому что в разных компаниях по разному устроены бизнес процессы.
Обычно выделяют:

- **Commit stage -** это стадия когда мы вытаскиваем код из системы контроля версий, его собираем. Проверяем статическими анализаторами, считаем разные метрики. Запускаем модульные тесты, которые проверяют, что на самом базовом уровне мы не облажались.
- Далее идет **Acceptance stage.** Это уже может быть задиплоидная в какой-то тестовой среде наша система, которая претерпевает разного рода тестирование. Например, разного рода сценариев, пользовательского интерфейса. То есть это некоторое, уже более длительное, чем на предыдущем шаге, автоматическое тестирование, которое проверяет уже некоторые более сложные варианты.
- Далее в данном случае идет распараллеливание на три этапа:
    - Диплоид этого всего на **нагрузочное тестирование**.
    - **Тестирование пользователем.**
    - **Production.**

Важно, чтобы все это перемещение проводилось одним и тем же скриптом. Очень плохо, если для каждого вида развертывания будет отдельный скрипт. Это значит, что в итоге будет не протестированный вариант размещение и то, что было сделано до этого потеряет весь смысл. Мы хотим, чтобы когда мы уже выкладывали в продакшен, в конфиге к скрипту менялся только одно число. Чтобы скрипт уже был много раз отлажен на других развертываниях.

Внизу картинки нарисована штука, которая называется **Artifact repository**. В нее идут стрелочки и из нее идут стрелочки, на которых написано **binaries.** Общая рекомендация по continuous delivery состоит в том, что бинарник версии собирается только один раз. Есть разные stage и для всех нужны бинарники, которые будут развертываться и проверяться. Было бы очень неразумно компилировать каждый раз бинарник. Если собирать каждый раз, то мы тратим дополнительно время на сборку и мы можем случайно собрать разные версии. Будет как-то глупо, что проверим мы одну версию, а в продакшен пойдет другая. И этот паплайн имеет смысл, только если мы проверяем один и тот же бинарник. Часто еще в современных системах continuous integration часто сохраняется бинарник, от него считается хеш и на каждой стадии хеш бинарника сравнивается. Потому что вдруг возникли какие-то ошибки при копирование или еще что. Есть специальные тулы для таких целей.

Полезные практики

- **Собираем бинарники только один раз**
- **Один и тот же процесс развёртывания для разных окружений.** Скрипт для deploy prodaction точно такой же как и для deploy testing. И только конфигурация этого скрипта меняется. Для этого мы очень хотим **отделить кода от конфигурации окружения.** То есть те вещи, которые меняются от тех вещей, которые не меняются. Это позволит нам один и тот же код диплоидь в разные места, меняя только конфигурацию.
- После того, как мы развернули бинарник на целевое окружение, всегда нужно **запустить smoke test.** Это простой тест, который просто запускает приложение и смотрит, что приложение запустилось, что все нужные компоненты проинициализированы, что появилось главное окно, которое показывает нужные текст, что база данных запущена. То есть простая проверка, что приложение поставилось и некоторым образом функционирует.
- Очень хорошо, если локально получится сделать **копию production окружения.** То есть есть сервера, которые будут в production и было бы хорошо, если бы такие сервера были локально.Очевидно, что тогда на этом окружение можно тренироваться, запускать тесты и так далее. Это не всегда может получиться. Например, это может быть дорого, потому что если все это работает на 200 серверах, то скорее всего вам не дадут дополнительные 200 серверов. Тогда можно попробовать это все сэмулировать виртуально. Если так тоже не получается, то можно попробовать сделать эмулятор. Если есть очень хитрое

оборудование, которого у вас просто нет, можно попробовать написать на него эмулятор, которые будет нужное окружение эмулировать. Это не даст никаких гарантий, но хотя бы что-то можно будет проверить.

- Еще один момент это **“кэширование” запусков конвейера.** Представьте, что есть 20 разработчиков и они регулярно коммитят, все как положено. Понятно, что несколько коммитов могут прийти до того, как закончится предыдущая сборка. И тогда логичнее запускать одну сборку для нескольких коммитов. Да, если сборка сломается, то нужно будет выяснять какой из коммитов именно сломал сборку, но это лучше, чем запускать сборку параллельно.

Создание конвейера развёртывания

По поводу создания конвейера. Если уже есть работающий проект, тогда что-то будет проще, что-то сложнее. Если нет ничего, то нужно быть готовым, что придется потратить достаточно много времени, чтобы все это настроить. Сейчас есть какое-то количество специальных тулов для этого, которые позволяют описывать work flow и задавать каким-то образом некоторые правила.

1. Сначала нужно **спроектировать основные этапы и создать скелет.** Если уже работа налажена, пишется код, тестируется, продукт выпускается, то нужно сходить к соответствующим людям и узнать как происходят все необходимые процессы, которые проходят и просто аккуратно их записать. Если новый проект, ничего совсем не написано. То можно начать очень просто. Есть три стадии: сборка, запуск тестов и релиз. Это можно сделать совсем просто. Написать hello world, написать тест, который всегда проходится. И пусть этот тест запускается, пусть генерируются отчеты. То есть система должна быть построена и должна работать, чтобы ее инкрементально можно было увеличивать.
2. **Автоматизирование сборки и развёртывания.** На этой стадии настраивается система сборки. Пишется какой-то код минимальный, пишутся тесты, делается так, что бы они запускались. Все это пару дней должно занять и делаться должно до того, как разработчик сел писать код. Если планируется новый проект, то нужно еще запланировать нулевую итерацию, на который вы будете заниматься структурными вопросами. Даже хотя бы git настроить, ide настроить, создать проект, настроить систему сборки, настроить систему тестирования. Это все занимает рабочий день, а если настраивать еще описанную ранее штуку, то это пару дней займет.
3. **Автоматизирование модульных тестов и анализа кода.**
4. **Автоматизирование приёмочных тестов**
5. **Эволюционирование конвейера**

Все это инкрементально может добавляться, это не обязательно все делать сразу. Если большой сложный проект, вероятно это и не получится. Хорошо, если получится автоматизировать хоть что-то. Очень важно считать метрику, за сколько строчек кода вы сможете донести то что вы сделали до пользователей. Если какие-то вещи не

автоматизируются, то их стоит описать, что бы все понимали когда это происходит и по каким событиям триггерам эта деятельность должна начинаться. Вероятно, это будет сложно сделать, если нет выделенного компьютера для этого. Потому что для этого нужна машина, с постоянным доступом к интернет. А промышленном проекте скорее всего найдется лишний компьютер для этого всего.

Модель зрелости процесса управления релизами

Practice	Build management and continuous integration	Environments and deployment	Release management and compliance	Testing	Data management	Configuration management
Level 3 - Optimizing: Focus on process improvement	Teams regularly meet to discuss integration problems and resolve them with automation, faster feedback, and better visibility.	All environments managed effectively. Provisioning fully automated. Virtualization used if applicable.	Operations and delivery teams regularly collaborate to manage risks and reduce cycle time.	Production rollbacks rare. Defects found and fixed immediately.	Release to release feedback loop of database performance and deployment process.	Regular validation that CM policy supports effective collaboration, rapid development, and auditable change management processes.
Level 2 - Quantitatively managed: Process measured and controlled	Build metrics gathered, made visible, and acted on. Builds are not left broken.	Orchestrated deployments managed. Release and rollback processes tested.	Environment and application health monitored and proactively managed. Cycle time monitored.	Quality metrics and trends tracked. Non functional requirements defined and measured.	Database upgrades and rollbacks tested with every deployment. Database performance monitored and optimized.	Developers check in to mainline at least once a day. Branching only used for releases.
Level 1 - Consistent: Automated processes applied across whole application lifecycle	Automated build and test cycle every time a change is committed. Dependencies managed. Re-use of scripts and tools.	Fully automated, self-service push-button process for deploying software. Same process to deploy to every environment.	Change management and approvals processes defined and enforced. Regulatory and compliance conditions met.	Automated unit and acceptance tests, the latter written with testers. Testing part of development process.	Database changes performed automatically as part of deployment process.	Libraries and dependencies managed. Version control usage policies determined by change management process.
Level 0 – Repeatable: Process documented and partly automated	Regular automated build and testing. Any build can be re-created from source control using automated process.	Automated deployment to some environments. Creation of new environments is cheap. All configuration externalized / versioned	Painful and infrequent, but reliable, releases. Limited traceability from requirements to release.	Automated tests written as part of story development.	Changes to databases done with automated scripts versioned with application.	Version control in use for everything required to recreate software: source code, configuration, build and deploy scripts, data migrations.
Level -1 – Regressive: processes unrepeatable, poorly controlled, and reactive	Manual processes for building software. No management of artifacts and reports.	Manual process for deploying software. Environment-specific binaries. Environments provisioned manually.	Infrequent and unreliable releases.	Manual testing after development.	Data migrations unversioned and performed manually.	Version control either not used, or check-ins happen infrequently.

Это модель зрелости компании. Здесь есть 6 параметров и для каждого из них есть 5 уровней начиная от -1 и заканчивая 3. -1 когда все очень плохо, 3 - когда все очень круто. И для каждого параметра можно оценить на какой стадии проект и сразу будет понятно к чему можно стремиться и на сколько это надо.

Что почитать