DEVOPS – ПРОЦЕССЫ И ИНСТРУМЕНТЫ V.1.2

devops-service.pro

- Ошибки при внедрении DevOps
- Регламенты взаимодействия с командой
- Ведение документации и DocOps
- Системы хранения исходного кода (SCM)
- Continuous Integration (CI). Непрерывная интеграция
- Continuous Delivery (CD). Непрерывное развертывание
- Infrastructure as code
- Автоматическое развертывание окружений
- Описание систем мониторинга
- Мониторинг производительности приложений (APM)
- Описание систем агрегации Log-ов
- Автоматизация тестирования
- Хранение артефактов сборок
- Интеграция тестирования безопасности - DevSecOps

https://devops-service.pro

+380(44)3641780

pustovit@devops-service.pro

andrewpas

https://www.facebook.com/devopskiev/

https://www.reddit.com/user/devops-service

https://devops-service.blogspot.com/

https://www.linkedin.com/in/andrew-pustovit-38687036/

https://medium.com/@devops_service

Об авторе

Меня зовут Андрей Пустовит, мне 45 лет. Живу в Украине, в Киеве. Около 20 лет посвятил ИТ и конкретно ИТ-аутсорсингу.

Сопровождал компании, организовывал и обучал службы поддержки. Хорошо знаю, как это надо делать и ещё лучше, как не надо.

Дал старт карьеры нескольким хорошим специалистам.

Пытался до конца верить в местный бизнес, в его зрелость в ИТ. В итоге, пришёл к парадоксальному выводу, что ИТ-услуги нужны исключительно ИТ-компаниям.

Сейчас я полностью сосредоточился на направлении **«DevOps as a service»**.

Для подробного и продуманного каталога услуг пришлось изучить запросы рынка и технические инструменты, которые эти запросы могут удовлетворить. Посмотреть, что вызывает у заказчиков скепсис и что им действительно может пригодиться и помочь.

Книга, или скорее реферат, который вы скачали – это мои попытки разобраться что такое концепция DevOps в разрезе инструментов и процессов. Какие осязаемые выгоды несёт её внедрение клиентам и командам.

Плюс «проговорить» на бумаге планшете свои мысли по поводу организации взаимодействия ИТ и бизнеса.

Документ будет периодически обновляться и, надеюсь, стает основой обучения для моих будущих сотрудников.

Я приглашаю «айтишников» добавляться ко мне в контакты и дискутировать.

Есть огромное количество вопросов, которые требуют обсуждения. Это в основном вопросы взаимодействия ИТ и всех остальных составляющих продукта. Очень интересен рынок ИТ-разработки, какое ваше мнение о развитии или стагнации.

Также я очень надеюсь на вашу обратную связь в письмах, на страницах в Facebook и LinkedIn.

https://www.facebook.com/devopskiev/

https://www.linkedin.com/in/andrew-pustovit-38687036

https://twitter.com/devops_pro

https://t.me/devops_service

andrewpas

pustovit@devops-service.pro

Вступление

DevOps уже не новое, но пока ещё модное явление в современных ИТ. Наблюдается большой дефицит профессиональных специалистов. Уже возник небольшой хаос в терминологии и начали появляться ответвления – DevSecOps, GitOps, ChatOps.

Общий тренд показывает, что DevOps-практики необходимо использовать, но не до конца ещё понятно зачем и как. Пока «потолок» применения DevOps – тотальная автоматизация всего и вся.

Но DevOps это больше о бизнесе и процессах, а не о серверах и исходном коде.

Раньше все было хорошо – разработчики пишут код, «админы» следят за доставкой кода, за сервисом и все относительно счастливы.

Но вот ИТ оказались в тренде и туда потекли рекой большие деньги. Ситуация стала меняться – массово исчезли старые добрые «разводняки» в виде «интересных проектов», «нужно на вчера», «работаем с тем, что есть».

«Бизнес» увидел, что ИТ это теперь дорогущий и сложный инструмент, которым надо не просто уметь пользоваться, а пользоваться максимально эффективно. В итоге затраты на ИТ увеличились, а результаты мягко говоря…

Понятно, что, например, сервис, который разрабатывается, это не только код и instance на AWS – это и капризные клиенты, маркетологи, исследования рынка, переговоры, привлечение денег, реклама (очень много рекламы) и прочие, неинтересные разработчикам и «админам» вещи.

DevOps и возник, как результат ответа на вопрос «Ребята, а можно как-то побыстрее шевелиться?» Оказалось, можно.

Поэтому главная задача DevOps – минимальный time to market (минимальное время вывода продукта на рынок). Поскольку уже есть большая конкуренция, избалованные пользователи и т.д. Надо бороться за выживание и развитие.

Остальные дискуссии (типа DevOps это Ops или Dev, должен писать код или нет) можно смело отбросить.

Появляется agile-цикл – выдали продукт, получили обратную связь от пользователей. Маркетологи с бизнесом её обработали – дали новые задачи для изменения продукта, разработчики выдали новую версию (версии)... и опять получили обратную связь.

Изначально практики DevOps лучше всего подходят сервисам – там вышеописанный процесс непрерывный. Но также они могут быть применены и к обычному бизнесу со значительной «ИТ составляющей» – внедрения, тестирования и обкатки, обновления, мониторинг.

Очень обобщённые задачи при внедрении DevOps-процессов:

- Обеспечить бизнесу максимально быстрый процесс проверки идей развития продукта
- Автоматизировать сборку и доставку кода на все окружения
- Автоматизировать тестирование
- Автоматизировать развёртывание окружений для разработчиков и production
- Раскрыть глаза разработчикам, где и с чем будет работать их код
- Помочь в разработке и изменении архитектуры сервиса
- Документировать свою работу, дать инструменты для документирования разработчикам
- Обеспечить подробный мониторинг

Человеческим языком задача DevOps сделать так, чтобы:

1. Разработчики (все, а не тот, который это придумал и знает, как работает) не боялись отправлять код в production, или разработать процедуру, при которой разработчики могут спокойно сделать merge в production-ветку, не боясь последствий; имеют возможность откатить назад без потерь данных и простоев сервиса.
2. Разработчики видели смысл в тестировании (вернее, чтобы разработчики начали писать тесты и воспринимали это как действительно полезный процесс, который приносит пользу).

3. Разработчики как можно быстрее получали то, что им надо и какие им надо окружения для тестирования и production.
4. Бизнес мог понимать, что вообще происходит – бизнес-метрики, сколько регистраций после первого посещения, время, затраченное на регистрацию и прочее, чтобы иметь материал с которым маркетингу работать дальше.
5. Кто-то (новый разработчик в команде, руководство, ПМ и т.д.) мог узнать, как вся эта конструкция из кода, серверов и сервисов работает (схемы, актуальная документация, сколько и за что платим и т.д.).

Это минимальные цели внедрения методологии.

Без их достижения «девопс» – это просто парень, который знает Linux, Zabbix и умеет писать скрипты, но получает за это неприлично много денег.

Чтобы такого «безобразия» не было надо чётко уяснить, что DevOps – это процессы, в какой форме они существуют не столь важно.

Руководство (если вы читаете этот документ, скорее всего вы и есть ~~сопротивление~~ руководство) и разработчики и остальной коллектив компании должны этот процесс понять и принять.

> Уже мало кто протестует против Scrum. Понятно, что большинство разработчиков протестует – они работать пришли, а не «стэндапить». Но если смотреть на «бизнес», то он очень даже не против. Вернее, ему все равно, но хорошо, когда есть какой-то порядок. А порядок всегда приводит к результату.
>
> Поэтому главная рекомендация – протестовать против DevOps тоже не продуктивно. Можно (~~и даже нужно~~) иронизировать, говорить, что это «очередное «не нужно» и хайп», но саботировать эти процессы точно не надо.

Может на начальном этапе старта проекта эта методология кажется излишней, но дальнейший рост без неё (или аналога, если его придумают) проблематичен, если вообще возможен в современных реалиях рынка.

Для внедрения DevOps всегда приходится использовать несколько инструментов, которые должны иметь возможность интегрироваться друг с другом. Иногда это довольно просто, иногда нет. Это все – потраченное время на изучение, настройку, тестирование и дальнейшую поддержку.

Это не сравнение характеристик и не подробное описание продуктов.
Мы верим, что наш читатель достаточно владеет вопросом, знает об описываемых инструментах и не нуждается в каком-то обучении основам. Это просто подбор проверенных средств для автоматизации, которые могут интегрироваться друг с другом.
Пока мы не освещаем Docker, AWS, Kubernetes и остальной «мэйнстрим». Это отдельная очень большая тема. И да – многие компании ещё этим не пользуются, «монолиты» все ещё живы и зарабатывают деньги владельцам сервисов.

> Итак, вы разработчики, у вас небольшая команда, все заняты, все пишут код. Все занимаются production в меру своих сил. Технический долг постепенно растёт, но не сильно.

Самое главное ваше желание – писать код, рабочий, красивый, чтобы не было стыдно (~~ну или потом переписать все на Go~~).

Есть понимание, что «что-то не так», или может активно смотрите по сторонам и видите этот хайп. И понимаете, что он не на ровном месте возник.

А может даже видите, что разработчики не пишут код, а изучают всякие «дженкинсы» и «гитлабы».

Значит надо “брать девопса”, или нанять разово чтобы он все вам настроил и рассказал.

Мы – DevOps-service.pro, дарим вам что-то вроде чек-листа или roadmap.

Мы постарались описать все значимые с нашей точки зрения процессы в DevOps методологии и привели список основных рабочих инструментов. Мы постараемся объяснить взаимосвязь процессов и их важность в разработке.

Документ бесплатен, но пока не открыт под open source лицензией и поэтому защищён от изменения и распечаток.

Книга специально сверстана для просмотра на мобильных устройствах, поэтому рекомендуем читать на планшете.

Успехов!!!

Технические процессы в DevOps

Под техническими процессами мы подразумеваем последовательность действий при разработке продукта, которые надо автоматизировать и улучшать.

Какие процессы мы бы выделили:

Работа с кодом. Хранение исходного кода продукта. Ролевая модель доступа. Взаимосвязь между проектами

6. Сборка кода. Компиляция из исходников, работа с зависимостями, тестирование
7. Доставка кода. Выгрузка артефактов сборки кода в инфраструктуру сервиса, откат изменений
8. Развёртывание окружений. Автоматическое создание новых тестовых окружений для разработчиков
9. Тестирование. Тестирование при сборке (unit-tests), нагрузочное тестирование, ручное тестирование
10. Мониторинг. Сбор метрик по производительности и загрузки ресурсов, сбор метрик по производительности самого приложения, визуализация.
11. Ведение и обработка «логов».
12. Документация. Составление и поддержка документации, вывод документов в разные форматы
13. Взаимодействие. Обучение, командная работа, регламенты
14. Безопасность. Интегрированный процесс анализа при разработке и тестировании.

Couds vs standalone?

Инфраструктура разработки. Своё или чужое? Брать готовый сервис или настраивать и поддерживать свой, на своих мощностях?

Наше мнение – начинать можно с облачных сервисов, если необходим приемлемый функционал и жалко (именно жалко) платить осознанные деньги за аренду «железки» в нормальном дата центре.

Потом, с дальнейшим ростом компании или проекта, это может «вылезти боком». Ну и дальнейшая миграция может быть болезненной и придётся немного всех переучивать.

Также бывают ситуации, проекты и заказчики, которые требуют использование именно своих серверов для всех процессов разработки и эксплуатации.

Рассмотрим более подробно плюсы и минусы каждого из вариантов:

CLOUD SERVICES	STANDALONE
ЦЕНА	
ЗА	**ЗА**
Изначально маленькая цена владения, расходы при росте компании прогнозируемы	Много компаний предлагают open source версии своих продуктов для установки на сервер, что позволяет использовать продукт бесплатно.

CLOUD SERVICES

ПРОТИВ

Со временем, при росте пользователей, ценник может неприятно удивить. Часто «маленькие» цены указаны при оплате на год вперёд. В бесплатных версиях урезанный функционал, «нарастить» который тоже стоит денег.

Вся бизнес-модель направлена на перевод клиента в сторону «среднего» платного тарифа. Цены могут меняться в одностороннем порядке

STANDALONE

ПРОТИВ

Часто open source версия имеет скудный функционал, полная версия стоит космических денег. При дальнейшем росте и переходе на платные продукты пользователи «докупаются» в больших количествах и может быть много лишних оплаченных лицензий. Часто необходимо оплачивать ежегодную подписку на поддержку и обновление продукта.

ФУНКЦИОНАЛ

ЗА

За относительно небольшие деньги на «средних» тарифах предлагается достаточно полный функционал, который покрывает большинство возможностей. Новые «фичи» добавляются для всех пользователей

ЗА

Как правило, присутствует достаточно много необходимых функций, платные поддержка и, как правило (но не удивительно), корпоративные «фичи» типа интеграции с AD или LDAP

ПРОТИВ

Функции почти всегда одинаковы для всех (нет возможности кастомизации), можно попасть в группу А/Б тестирования и получить баги в самый неподходящий момент. Интеграция с другими продуктами может носить «политический» характер или иметь технические ограничения. Наращивание функций может быть в виде интеграции с другими облачными сервисами, которые тоже не бесплатны.

ПРОТИВ

Часто для продуктивной работы необходимы большинство функций платной версии.

Хотя сейчас пошёл тренд на включение «вкусных» функций в open source версии с небольшим опозданием

ПРОИЗВОДИТЕЛЬНОСТЬ

ЗА

Хватает для большинства стандартных задач.

ЗА

Ограничено исключительно мощностями сервера. Если не хватает, можно арендовать дополнительные сервера.

ПРОТИВ

Как правило, задаются лимиты в каждом конкретном пакете. Instance выдаются «усреднённые» поэтому при пиковых нагрузках в проекте придётся переходить на новые пакеты или оплачивать увеличение мощности

ПРОТИВ

При неоптимальном использовании мощности в основном простаивают

НАДЁЖНОСТЬ

ЗА

Все вопросы по функционированию сервиса лежат на плечах владельцев сервиса. Клиенты вообще не заморачиваются этим. Как правило, надёжны.

ЗА

При правильном подходе, кадрах и хостере достаточно надёжен. Можно сервер даже в офисе поставить, чтобы не зависеть от провайдера

CLOUD SERVICES	STANDALONE
ПРОТИВ	**ПРОТИВ**
Сервисы все же «ломаются», причём поломка носит катастрофических и почти глобальный характер. Как классический пример, «убийство» базы данных в GitLab на production.	Все расходы и риски лежат на компании

ДОЛГОВЕЧНОСТЬ

ЗА	**ЗА**
Если сервис не «умер» в первые три года, значит будет жить дальше с переменным успехом и получать раунды финансирования, а может даже зарабатывать деньги.	При построенных процессах и настроенном софте может работать вечно. 1С версии 7 до сих пор жива и используется огромных количеством компаний (как пример из реальной жизни)
ПРОТИВ	**ПРОТИВ**
Может быть продан, поглощён и т.д. Что будет с продуктом, функциями, ценами и т.д. решает рынок и новый владелец. Может вообще закрыть как убыточный	Риски аналогичные облачным сервисом. Рынок, поглощения, новые владельцы, сворачивание

БЕЗОПАСНОСТЬ

ЗА	**ЗА**
Используются стандартные лучшие практики, в большинстве случаев есть двухфакторная аутентификация и т.д.	Внедряется самостоятельно администратором, плюс вендор даёт возможность дополнительных опций (та же двухфакторная аутентификация)
ПРОТИВ	**ПРОТИВ**
Поскольку сервис, как правило, создаётся «слоями», архитектурно может быть проблематично перейти на лучшие практики или обновлять алгоритм, не затрагивая пользователей безболезненно	Огромный риск человеческого фактора или недостаточной подготовки специалиста

Плюсы и минусы одинаковы для всех решений. Дальше мы будем ориентироваться исключительно на standalone-решения.

Работа с исходным кодом

«Ветвление» Git

Это начало всего. Команда должна чётко решить, как и по каким схемам будет идти ветвление, и соответственно автоматические сборка и развёртывание.

Есть как минимум 3 классифицированных наукой и ~~зоологами~~ разработчиками вида ветвлений и также самостоятельные изобретения команд и отдельных умельцев.

Поскольку требуется полностью автоматическая работа с ветками в плане сборки и развёртывания, от выбора типа ветвления зависят соответствующие инструменты.

Инструмент для сборки и доставки в основном «задаёт» пользователю один вопрос – «скажи мне какую ветку или тэг собирать и куда выкладывать».

С одной стороны, вроде легко – «ветку dev выкладывать на окружение dev.company.com». В большинстве случаев тестировщику или команде надо зайти на https://dev.company.com и посмотреть, что там и как.

С другой стороны, а если ветка явно не называется dev? Она может называться

feature-{some feature what mean developer or Jira ticket ID}.

И в этом случае не все системы могут понять, что от них хотят, поскольку они не умеют работать с динамическими именами веток. А по логике DevOps, разработчик должен иметь возможность развёртывания и теста ветки.

А как быть, когда несколько связанных между собой репозиториев? И у каждого своя система ветвления? Как говорилось выше – DevOps это процесс, соответственно нужна унификация. И её надо сделать, чтобы двигаться дальше.

Когда мы перестанем быть такими жадными, то выложим на сайт статью из внутренней базы знаний про ветвления в Git, а пока даём информацию для самостоятельного изучения, ключевые слова в поиске: **GitFlow, GitLab Flow, GitHub Flow.**

Хранение и администрирование кода

Сервер, на котором хранится исходный код. Основные минимальные функции, которые от него требуются:

Собственно, хранение и бэкап исходного кода.

15. Доступ к коду с аутентификацией (кто это?) и авторизацией (кто это мы поняли, теперь надо выяснить, какие у него права по работе с системой) по паролю и ssh ключам
16. Pull requests
17. Возможность клонировать репозитории по ssh и https
18. Возможность подписывать код PGP-подписями разработчиков
19. Обсфуцировать или «прятать» конфиденциальные данные

Мы опишем исключительно функционал хранения и доступа к коду, поскольку полный обзор каждого продукта займёт много времени.

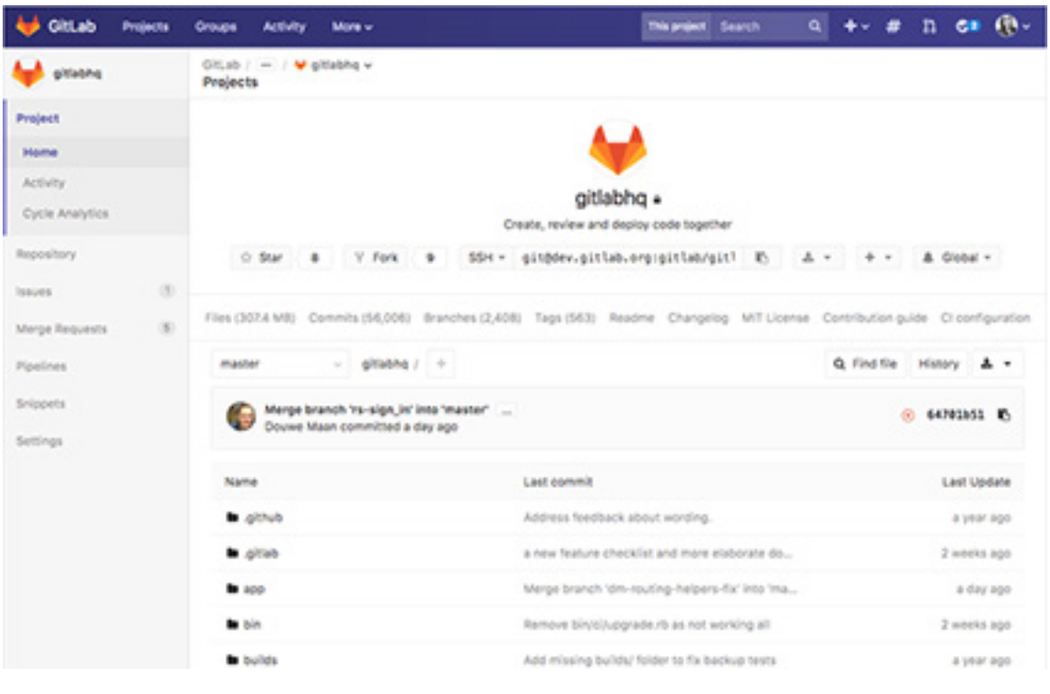
GitLab

Сайт, документация, дополнение для VSCode, дополнения для JetBrain IDEs.

Продукт одноименной компании, поддерживает GitLabFlow (они его придумали и делают продукт на основе этого Flow).

Позиционируется разработчиками как «всё в одном» для комфортной разработки. Есть облачная версия и версия для установки на свой сервер.

Написан в основном на Ruby, в качестве бэкенда использует PostgreSQL. Доступ для администрирования и работы через web-интерфейс. Работает только с Git.



Поддерживает двухфакторную аутентификацию и интегрируется с Active Directory и OpenLDAP.

Поддерживает расширение Git LFS для хранения и работы с большими файлами.

Интегрируется с Jira и Slack через web hooks.

Есть возможность ограничивать доступ к конкретным веткам проекта – Protected Branches.

Поддерживается Mirroring repositories – зеркалирование репозиториев между несколькими серверами Git. В бесплатной версии поддерживается только push в другой репозиторий. Имеет несколько пакетов подписки на on-premise версии, которые достаточно сильно отличаются по поддерживаемым возможностям. Имеет все предпосылки стать лидером в своей нише.

Также одна из сильных сторон этого продукта – встроенный сервер хранения артефактов сборки, начиная от бинарных файлов и заканчивая образами Docker.

Bitbacket

Сайт, документация, плагины, дополнение для VSCode, дополнения для JetBrain IDEs.

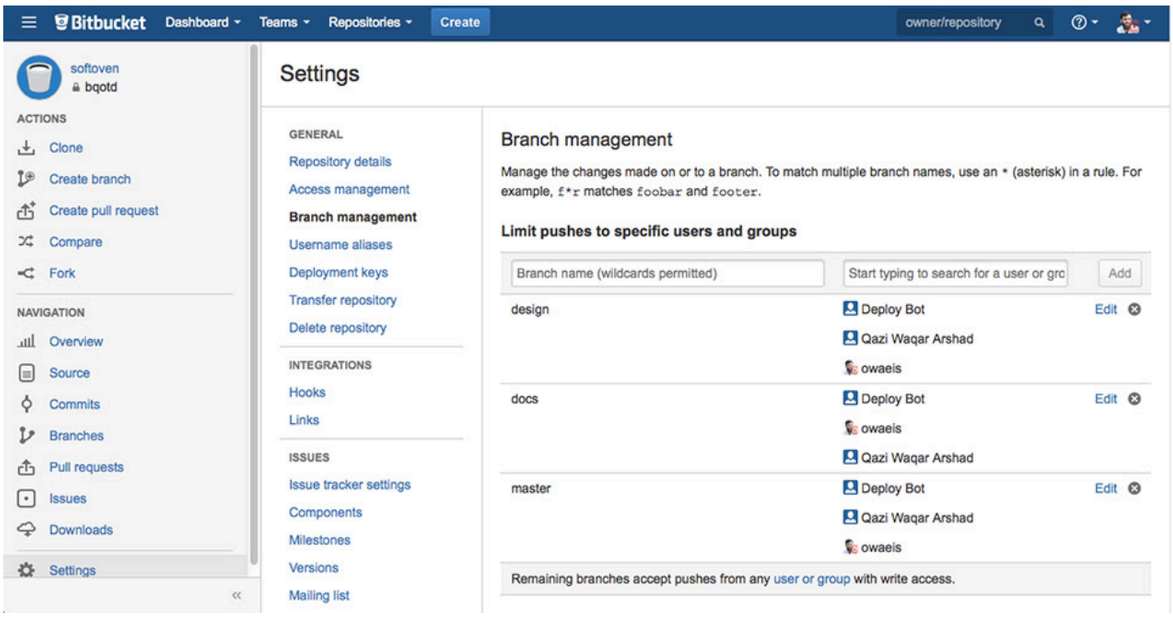
Продукт компании Atlassian, отлично поддерживает GitFlow, весь функционал «заточен» в основном под него.

Прекрасно интегрируется с другими продуктами Atlassian.

Написан на Java, в качестве бэкенда используется PostgreSQL (вернее предпочтительней всего его использовать). Работает только с Git

- Аналогичные возможности как у GitLab:
- Двухфакторная аутентификация
- Поддержка Webhooks
- Branch permissions – ограничение доступа к определённой ветке
- Аутентификация через Active Directory, LDAP, Jira Crowd

Естественно отличная интеграция со стеком Atlassian – Jira, Slack, Bamboo и Confluence.



Continuous integration и Continuous delivery

Это основополагающий принцип и одна из целей трансформации DevOps.

В основе этого принципа лежит понятие конвейера (pipeline) – все этапы после разработки: тестирование, сборка, доставка результатов сборки проходят автоматически, в строгой очерёдности с минимальным вмешательством людей. Что позволяет в разы сократить время доставки продукта потребителям и при этом обеспечить качество и безопасность.

Фактически состоит из двух процессов – непрерывной интеграции (Continuous Integration – CI) и непрерывной доставки (Continuous Delivery – CD)

Continuous Integration (CI)

Непрерывная интеграция (Continuous Integration) – это процесс постоянного слияния рабочих копий кода и сборки и тестирования проекта.

Преследует самую главную цель – недопустимость регрессий кода.

Регрессия кода.

Это явление простыми словами можно описать так:

на ранней стадии разработки допущена ошибка в модуле, но её пока не обнаружили

- от этого модуля зависят многие другие
- ошибка попадает в рабочую ветку
- модуль с ошибкой клонируется всеми разработчиками к себе на машины
- все работают долгое время без слияния, тестируя сборку у себя на компьютерах, ошибка не проявляется
- перед стартом или тестированием начинаются слияния веток разработчиков в проект
- сборка, запуск, ошибка сборки или функционала
- начинают разбираться, что случилось, кто виноват
- находят ошибку в модуле, делают задачу разработчику
- разработчик исправляет ошибку, но остальным приходится переделывать свои модули, которые зависят от первоначального
- при долгой разработке без слияний и тестов убрать регрессию все сложнее и сложнее, особенно если ошибок много, и они архитектурные

Это напоминает конвейерный принцип работы: «запорота» деталь – конвейер останавливают и разбираются в чем проблема.

Раньше интеграция веток кода разработчиков в основную была как один из завершающих этапов, а потом все разбирались почему проблемы и продукт не работает.

Один из принципов работы с Continuous Integration это групповая «починка» «поломанной» сборки. Как правило, это справедливо для сложных проектов с запутанной, монолитной архитектурой.

Естественно, что не начинают мигать красные лампы, и сирена не включается, когда сборка «ломается» каким-то коммитом. Просто CI сервер() GitLab, Bitbacket) может быть настроен на разные политики работы в таких ситуациях.

Можно запретить принимать другие коммиты пока не пройдены тесты и сборка на ветке, которую хотят слить с общим деревом. Так называемые gated commits – шаблон, в котором коммит не сливают с веткой, пока он не пройдёт тесты или успешную сборку.

Можно настроить обязательное code review от нескольких разработчиков перед слиянием.

Схема Continuous Integration очень сильно зависит от системы ветвлений Git (об этом говорилось выше).

Принципы Continuous Integration

Нет никаких особых правил по построению CI, каждая команда уникальна и имеет свои процессы.

Но можно выделить несколько общих вопросов, на которые стоит обратить внимание:

Простое частое слияние без тестов в основную ветку ничего не даёт само по себе. Кроме возможного «мусора» в ветке и последующей ошибки

- Необходимо очень точно рассчитывать мощности для CI/CD. Возможны ситуации, когда будет идти одновременно несколько сборок и тестов. В зависимости от технологий и архитектуры одна сборка может занимать достаточно много времени. Остальные в этот момент будут стоять в очереди. Надо планировать сразу несколько workers на серверах чтобы можно было параллельно работать. Поэтому не стоит жалеть средства на сервера для сборки и тестирования. Можно также настроить работу в виде кластера.

- Очень распространено понятие «ночной сборки» (nightly build). Слияние и сборка происходят в нерабочее время чтобы не загружать мощности днём.
- CI это не только поиск ошибок – это ещё и тестирование идей бизнеса, да и вообще всего рабочего процесса. На каждой такой короткой итерации: Слияние -> Тестирование -> Сборка -> Доставка в кружение для тестов -> Приёмочное тестирование; все ответственные сотрудники, начиная от QA и заканчивая владельцем продукта постоянно добавляют по мелочам определённую ценность продукту. Где-то найдена ошибка, где-то опечатка, где-то решили поменять концепцию или дизайн, или бизнес-логику.
- У команды должен быть понятный, принятый всеми алгоритм поведения в ситуациях, когда «ломается» сборка проекта

Continuous Delivery (CD)

Это логическое продолжение процесса CI (Continuous Integration).

Continuous Delivery (Непрерывная доставка) – это практика разработки программного обеспечения, когда при любых изменениях в программном коде выполняется автоматическая сборка, тестирование и подготовка к окончательному выпуску.

Главная цель CD – master-ветка (в большинстве вариантов ветвления) всегда должна быть готова к работе на production-окружении.

В разделах «Тестирование» и «DevSecOps» также есть важные моменты по сборке: проверка на ошибки, проверка зависимостей, проверка уязвимостей в зависимостях и проверка лицензий в зависимостях (это бывает очень важно при заказной разработке или при продаже компаний или сервисов).

Какой общий процесс сборки кода и доставки его результатов (в терминологии CI/CD это называется **артефакт**) на сервера:

1. Код выкладывается в репозиторий
2. Система сборки следит за изменениями в репозитории. Если появляется новая версия, которая соответствует определённым триггерам или их комбинациям: имя ветки, текст в коммите, тэг – начинается процесс сборки. Как именно производится сборка зависит от используемых в проекте технологий и фреймворков.
3. Сборка может производиться на самом сервере, во временных образах Docker, на других физических или виртуальных серверах с помощью агентов. Естественно там должны быть все инструменты и библиотеки для сборки конкретного кода.
4. Могут выполняться определённые действия перед сборкой и после сборки. Проверка кода на безопасность или корректность. Прогон unit-тестов. В зависимости от результата сборка может не запуститься.
5. Потом начинается процесс доставки на сервер или сервера. Это может быть сделано в автоматическом режиме или в ручном. Самая простая схема доставки, следующая (пример – приложение на Node.js в качестве бэкенда у web-сервера). Останавливается web-сервер, останавливается процесс Node.js, из директории сервиса удаляется старый код, с сервера сборки копируется артефакт, запускается процесс Node.js с новым кодом, запускается web-сервер.

Может случиться, что новый код работает не корректно и надо «откатиться» на предыдущую версию. Тогда аналогичный процесс сборки и доставки проводят с предыдущим рабочим коммитом в проекте.

Это самое простое описание процесса. Как вы понимаете, в реальной жизни полная остановка сервисов может быть на тестовых окружениях. На production ситуация совершенно другая. Серверов несколько, они стоят за load-balancer, доставка кода происходит по очереди.

Какие характеристики инструмента необходимы для сборки и доставки кода на сервера:

Интеграция с системами проверки и тестирования

6. Поддержка агентов и распределённых сборок

7. Поддержка разных VCS (Version Control Systems) – Git, Mercurial, SVN
8. Pre и Post скрипты
9. Поддержка мульти проектной сборки (в самых сложных случаях)
10. Поддержка использования SSH в сценариях (как минимум, должны же мы как-то попадать на удалённые сервера)
11. Возможность отправлять результаты сборки в различные мессенджеры, email и т.д.
12. Возможность автоматического или ручного отката deploy
13. Маскировка конфиденциальной информации (пароли, API-ключи) в скриптах и логах сборки

Инструменты CI/CD

Самые распространённые: Jenkins, Bamboo, GitLab CI, TeamCity.

Jenkins, TeamCity и Bamboo самостоятельные продукты, GitLab CI является логической частью GitLab, использует отдельный модуль GitLab Runner, который может использоваться как самостоятельное приложение.

> Фаворитом является Jenkins. Главная особенность – прекрасная работа с multi pipeline (взаимосвязанная сборка по очереди нескольких проектов), Bamboo также имеет такой функционал (build plan). GitLab позволяет это сделать в виде зависимостей при сборке одного проекта, что достаточно «костыльно».

Jenkins

Сайт, документация, плагины

Сам по себе он имеет достаточно скудный функционал, но расширяется с помощью плагинов. Как и любое «плагинообразное» произведение имеет как плюсы, так и минусы.

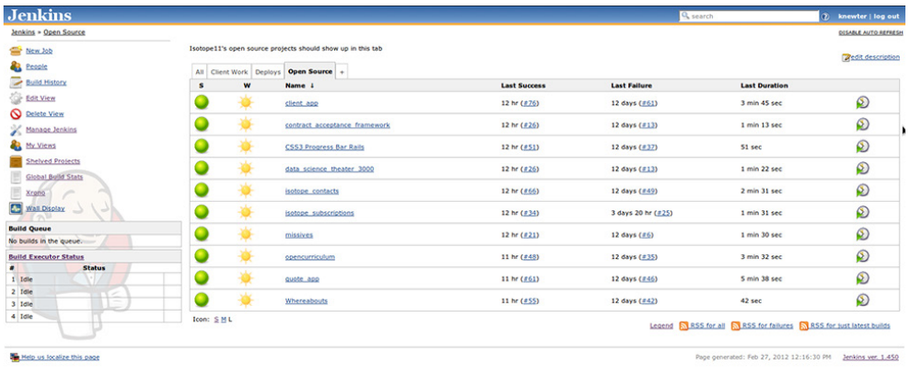
Плюсы – бесконечные возможности расширения, бесплатно в большинстве случаев, open source, пока ещё есть интерес к продукту (да и сам продукт, при наличии сильных конкурентов тоже развивается).

Минусы – зоопарк плагинов, которые надо отслеживать и обновлять, изучать; некоторые важные плагины заброшены авторами и нет форков.

«Наскоком» из него выжать приемлемый результат будет трудно, но при приложении определённых усилий получается хороший инструмент.

Написан на Java. Сборка настраивается через графический интерфейс или через специальный файл jenkinsfile (такой функционал требует установки соответствующего плагина).

С помощью Web-hook можно интегрировать со Slack, email. Поддерживает распро-



странённые VCS. Поддерживает работу через распределённых агентов. Полностью бесплатен.

Есть форк Jenkins X – https://jenkins-x.io, который предназначен для работы с Kubernetes.

Bamboo

Сайт, документация, плагины.

Продукт компании Atlassian, что подразумевает отличную совместимость с остальными продуктами компании. Задания описываются с помощью yaml. Поддерживает работу через агентов. Большинство плагинов платные. Агенты платные.

GitLab CI (GitLab Runner)

Сайт, документация.

Поддерживает агентов, может собирать проекты как на голом «железе», так и в Docker образе. Есть встроенная поддержка доставки в Kubernetes. Задания описываются на yaml.

Из поддерживаемых VCS только Git. Есть возможность rollback (отката) на предыдущую сборку. Через Web-hook интегрируется с Jira, Slack, email. Откат на предыдущую версию поддерживается в явном виде. Есть возможность создавать отдельные экземпляры «сборщиков» (runners) для каждого проекта, или использовать один «сборщик» на группу проектов.

Есть возможность импорта или шаблонизации «раннеров» с доугих проектов.

Можно вставлять credentials для доступа к серверам, как переменніе в проекте.

«Сборщик» (runner) описывается на языке TOML.

TeamCity

Сайт, документация, плагины.

JetBrains выпускает отличные продукты, которые становятся стандартном де факто в разработке.

Мощный, гибкий продукт. Сборки описываются в виде кода на языке Kotlin, который также разработали в JetBrains. Стандартная схема лицензирования с сервером и агентами – «играемся» с тремя агентами и учимся бесплатно, работаем в production уже за приличные деньги.

Очень рекомендуется тем командам, у которых весь процесс построен на стеке продуктов JetBrains, как вы понимаете там отличная совместимость и интеграция с TeamCity.

Можно интегрировать TeamCity с любой Jetbrains IDE и запускать сборку на сервере сразу после локальных коммитов на машине разработчика.

Благодаря необходимости изучения Kotlin у TeamCity достаточно высокий порог вхождения.

Bamboo, GitLab CI целостные продукты, которые предлагают описывать процесс сборки и доставки на yaml. Грубо говоря описание этапов, подстановка переменных, и связка SSH плюс bash скрипты. В итоге получается достаточно гибкое решение.

Главная особенность всех этих продуктов – полезные возможности по безопасности.

«Прятать» пароли в основном интерфейсе в виде ключ-значение, потом ключ вставляется в план сборки и развёртывания в виде переменной, можно спрятать вывод паролей в log-ах.

Если у вас простой проект в виде «собрали, протестировали, выложили на сервер» – тогда подходит GitLab CI. Если что-то сложнее, когда несколько модулей зависят друг от друга – тогда Jenkins, TeamCity и Bamboo. Если необходима максимальная интеграция с распространёнными системами тестирования и прочим – тогда Jenkins.

Следующий момент – подключение дополнительных ресурсов для сборки и тестов. Схема такая – есть управляющий сервер и есть условные агенты, которые выступают в виде мини-кластера. Центральный сервер при больших нагрузках и сборки разных проектов начинает использовать мощности агентов. Соответственно «тяжёлый» проект собирается на несколько порядков быстрее.

В данной ситуации Jenkins и GitLab CI позволяют бесплатно добавлять любое количество дополнительных мощностей. Bamboo и TeamCity (3 агента бесплатно) продают дополнительных агентов за немалые деньги.

Особо хитрые и бережливые выделяют под них очень мощный сервер, который с головой покрывает текущие потребности по мощностям для сборки.

Методики доставки артефактов сборки

Выше в этой главе мы описали самый простой процесс доставки собранных артефактов на сервер.

Опишем подробней основные алгоритмы работы непрерывной доставки. Основная цель этих паттернов – не останавливать работающий сервис, или останавливать таким образом, чтобы работающие пользователи сервиса ничего не заметили и не потеряли свои данные.

Термин «окружение» (environment) – совокупность серверов, системных библиотек, операционных систем, необходимого ПО, вспомогательных сервисов (базы данных и т.д.), сетевой модели (маршрутизаторы, DNS, load-balancers и т.д.), которые используются в проекте.

Окружение описывается в виде кода (см. «Развёртывание окружений») и **ОДИНАКОВОЕ НА ВСЕХ** площадках.

При внесении изменений в параметры окружений, они **ПЕРЕСОЗДАЮТСЯ НА ВСЕХ** площадках.

По крайнем мере, так должно быть в идеале. Нельзя иметь в одном окружении CentOS 6, а на другом CentOS 7. Это утрированно, но смысл должен быть понятен.

В разработке используются как минимум три окружения, с соответствующими ветками в Git.

В каждой команде или проекте они могут называться по-разному, но от этого их назначение не меняется.

Окружение dev (dev-ветка) – самое первое место для тестирования. Весь «сырой» код вначале доставляется на это окружение и идёт самое подробное тестирование.

Окружение stage (stage-ветка), копия базы данных с production – сюда уже идёт код, который разработчики готовы отдавать в production. Опять этап тестирования.

Окружение production (master-ветка) – код из этой ветки будет идти в наш рабочий сервис.

Различают два основных вида доставки, между которыми есть принципиальные отличия.

Развёртывание – установка конкретной версии ПО в конкретной среде. У нас есть условно версия1.0 и production окружение. В пределах этой версии устанавливаются какие-то доработанные модули, багфиксы и т.д. Основной функционал продукта остаётся неизменным.

Релиз – создание новых функциональных возможностей продукта или сервиса. Иными словами, версия 2.0 приложения, в котором есть или принципиальные отличия, или новые функции. В этом варианте можно много чего поломать, начиная от баз данных (меняется структура, добавляются новые таблицы) и заканчивая оттоком пользователей, которые не восприняли нововведения, простои после поломок, потерю своих данных.

Соответственно надо продумать систему доставки релизов, которая избавит нас от таких рисков.

Пока разработаны две стратегии – **релиз на основе среды исполнения** и **релиз на основе приложения**.

Релизы на основе среды исполнения.

Релиз на основе среды – используется несколько окружений и только одно получает настоящий трафик.

В этом случае релиз развёртывается только на одно окружение, на которое перенаправляется настоящий трафик. Виды релизов на основе среды исполнения – **blue-green deploy**, **canary deploy**, **cluster immune systems**.

Blue-green deploy.

Есть две производственные среды – **«синяя»** и **«зелёная»**. В любое время только одна обслуживает трафик. Новая версия разворачивается в неактивную среду. Потом на неё переключается пользовательский трафик. Но при этом возникает проблема базы данных бэкенда.

Есть два подхода для решения этой проблемы:

- Создание двух баз данных («синяя» и «зелёная») для соответствующей версии среды. Во время релиза синяя база переводится в режим read-only, копируется в «зелёную» и на неё же переключается трафик. Откат возможен при переносе diff из «зелёной» в синюю базу.
- Отделение изменений базы данных от изменений приложения. К базе делаются только дополнения, но существующие объекты никогда не изменяются. Также в приложение не «вшиты» версия и тип базы.

Подобную ситуацию с базами данных можно отнести больше к SQL базам. NoSQL базы, типа MongoDB, спокойно относятся к такого рода доставки артефактов, поскольку могут обрабатывать данные любой структуры в json-формате.

Как видите, если проект большой и нагруженный, потребуются большие вложения в инфраструктуру, особенно для миграции SQL баз.

Канареечные релизы и cluster immune systems

Канареечный релиз – развёртывание на все более крупные и на более критически важные среды по мере уверенности, что код работает. От малого большому.

Под «удар» вначале попадают тестировщики, сотрудники и т.д. Наиболее известный тип таких развертываний проводят Facebook и Google, постепенно внедряя новые версии.

Cluster immune systems – расширение шаблона канареечного релиза, в ней системы мониторинга связываются с процессами релиза и автоматизируется откат на предыдущую версию кода, если производительность системы со стороны пользователей отклоняется от заданных показателей сильнее чем ожидается.

Но это уже высший пилотаж. Похожий принцип деплоя есть «из коробки» в Kubernetes.

Релиз на основе приложения

Такую систему доставки приложений надо «уметь готовить» и заниматься этим с самого старта проекта.

Этот алгоритм реализуется с помощью переключателей функциональности **Feature Toggling** или при использовании веток функциональности **Feature Branch**.

Feature Branch – создаётся отдельная ветка с Feature (какой-то новой возможностью). Разработчик над ней работает. Писать код в эту ветку он может довольно длительное время. Когда работа закончена ветка попадает в master.

При этом разработчик может работать параллельно и над общей веткой. Таким образом накапливается технический долг (см. Continuous Integration). С учётом того, что ветка не попадает в процесс непрерывной интеграции, то потом, в случае ошибок и зависимостей от основной ветки может быть регресс приложения или сервиса.

Поэтому появился алгоритм **Feature Toggling** – с помощью специальных библиотек в коде добавляются флаги «фичи», которые можно включать и выключать, если что-то пошло не так.

Флаги можно маскировать, но при этом ветки будут все равно нормально сохраняться в мастер-ветке, не нарушая процесс CI/CD.

Подробнее про Feature Flagging можно почитать в специальном документе Feature Flagging Guide.

Уже есть даже стиль разработки – Feature Flag-Driven Development. Хорошая обзорная статья на эту тему.

Библиотеки, которые можно использовать в коде для Feature Toggling:

- iOS/Android Feature Flags
- Go Feature Flags
- Java Feature Flags
- JavaScript Feature Flags
- .NET Feature Flags
- Node JS Feature Flags
- PHP Feature Flags
- Python Feature Flags
- Ruby Feature Flags

Хранение артефактов сборки

Бывают проекты, результатом сборки которых есть простой файл. Неважно какого формата: rpm, deb, msi и т.д. Иногда приложение может быть кроссплатформенным и поддерживать несколько типов операционных систем. Возникает потребность в хранении результатов сборки.

Можно настроить анонимный FTP-сервер и выкладывать артефакты на него. Это не совсем удобно, да и не современно. Ведь надо хранить несколько версий, некоторые версии (например, «ночные» сборки), должны быть доступны не для всех, а для тестировщиков.

Может понадобиться сделать приватный репозиторий для rpm, deb, которые будут доступны по подписке для клиентов и доступ будет по лицензионному ключу. Надо делать описание и changelog.

Если используется концепция DevSecOps, то может понадобиться поддержка своего репозитория deb или rpm, которые будут использоваться как зависимости при CI/CD. Короче, есть потребность в своём сервере хранения и дистрибуции артефактов сборок.

Есть отдельный класс ПО, который решает эти задачи.

Наиболее известные это:

- Nexus Repository Manager (бесплатная версия)
- JFrog Artifactory (бесплатная версия).
- GitLab Package Registry

Nexus Repository Manager

Сайт, документация, плагины.

Поддерживает следующие форматы репозиториев: Apt, Docker Registry, Git LFS Repositories, Go, Maven, Npm, NuGet (Windows пакеты exe, msi), PyPI, RubyGems, Yum.

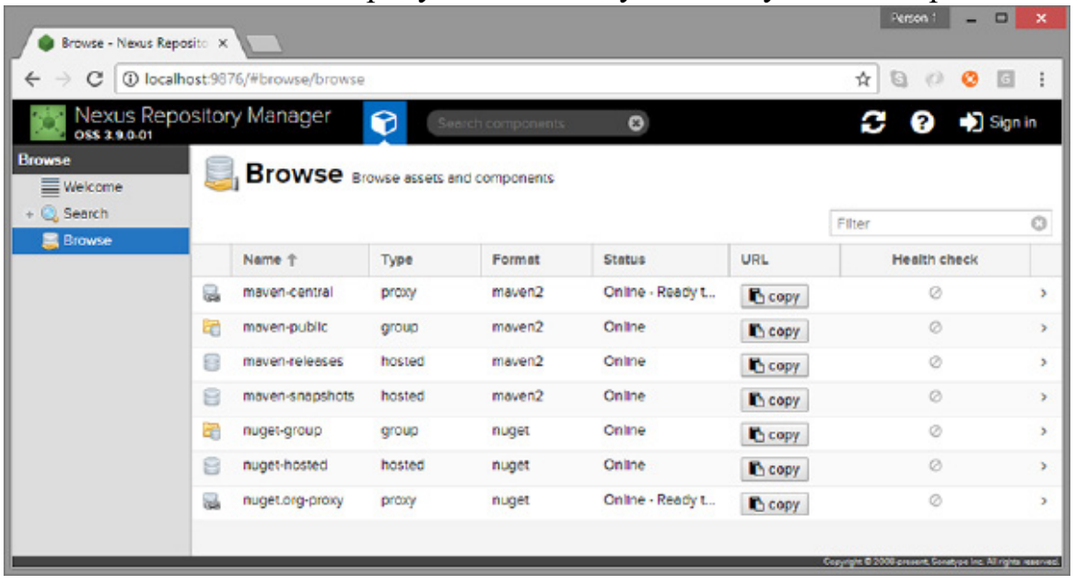
Можно объединять репозитории в группы. Позволяет интегрироваться с CI/CD для прямой публикации артефакта сборки прямо в репозиторий.

Поддерживает ролевую модель доступа к репозиторию, как к интерфейсу управления, так и к доступу через пакетные менеджеры.

Иногда бывает ситуация, когда недоступны некоторые публичные репозитории типа Npm, модули из которых используются как в зависимости для проектов, и вся сборка останавливается.

Можно настроить Nexus Repository Manager в качестве proxy, который будет хранить на себе версии нужных зависимостей и при этом будет периодически обновлять на новые версии (сохраняя старые).

Очень мощный и полезный продукт. Рекомендуется к изучению и применению.

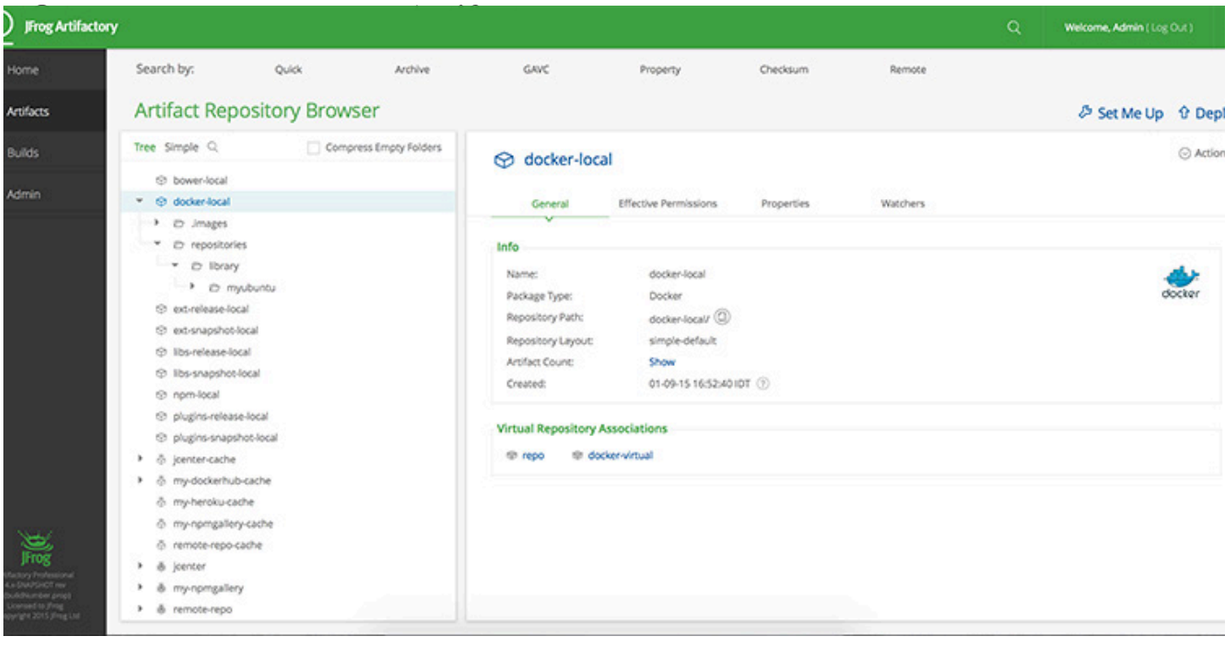


JFrog Artifactory

Сайт, документация, плагины и способы интеграции.

Очень качественный коммерческий продукт. Есть Open Source версия, которая поддерживает только артефакты сборок Maven, Gradle, Ivy. Которые через плагины могут собрать rpm, deb, npm. Можно пробовать и экспериментировать.

ТакжеJFrog выпустили бесплатную версию JFrog Container Registry – сервер для частного реестра для хранения Docker-образов и Helm charts.



При огромных возможностях коммерческой версии цены достаточно ощутимые.

Больше подходят для зрелых компаний и проектов.

Недавно был выпущен JFrog Container Registry – бесплатное хранилище приватных Docker репозиториев, который можно установить на свой сервер.

GitLab Package Registry

Сайт, документация.

Входит в состав продутка Gitlab, может быть развернут на своем сервере. Есть хороший API для работы через скрипты. В бесплатной версии практически отсутствует возможность ограничения доступа к артефактам.

Может выступать репозиторием для Composer, Conan, Go, Maven, NPM, NuGet, PyPI, Generic packages.

Тестирование

Можно написать много «простыней» текста, что тестировать надо, что это качество продукта, новые, довольные пользователи и т.д. Мы думаем это писать не надо. Это и так все понимают.

Мы опишем основные методики тестирования и продукты, которые позволяют автоматизировать тестирование. Также рассмотрим пару инструментов для нагрузочного тестирования.

Методики тестирования

Если хочется подробностей – вот прекрасные документы на эту тему:

1. https://dou.ua/forums/topic/13389/
2. http://www.protesting.ru/testing/testtypes.html

Очень много инструментов по тестированию безопасности продукта или сервиса описаны в главе «DevSecOps».

Сразу можно сказать, что полностью автоматизировать тестирование невозможно.

Нужен человек под названием Manual QA, который вооружён методикой, чек-листами и терпением.

Сейчас набирает популярность подход «разработка через тестирование» (**TDD –Test Driven Development**) – вначале пишутся тесты, потом код.

Также есть ответвление «Разработка через приёмочные тесты» – **Acceptance test driven development (ATDD)**. Общий смысл в том, что прежде чем что-то делать, надо придумать критерий выполненной работы и критерий того, что работа сделана правильно. В общем долго, дорого, надёжно.

Что хочется описать более подробно из всего этого многообразия. И как это можно «запихнуть» в процессы автоматизации всего и вся.

Модульное тестирование, или юнит-тестирование (англ. unit testing) – позволяющий проверить на корректность отдельные модули исходного кода программы, используя «заглушки».

Например, нам не надо делать запрос на авторизацию, мы просто пишем «заглушку», которая возвращает правильные значения (типа мы авторизовались) на логин “user” и пароль “12345”, и дальше идёт сам тест. Для модульного тестирования надо, во-первых, уметь и хотеть писать тесты, во-вторых использовать специальные библиотеки или фреймворки.

Вот список фреймворков для модульных тестов. Выбираем на любой вкус и цвет, под любой язык программирования.

Как можно внедрить поддержку автоматического модульного тестирования в CI/CD?

Очень просто – задача вначале протестировать и в случае успешного прохождения теста запустить сборку. Для этого есть pre-build скрипты в любом из рассматриваемых

инструментов в главе «Continuous Delivery (Непрерывная доставка)».

Функциональное тестирование – тестирование всех функций продукта в целом. Как правило, выполняется вручную по чек-листам. В основном таким образом тестируются web-приложения и сервисы.

Есть определённый ряд инструментов, которые автоматически проводят функциональное тестирование.

Принцип следующий. Устанавливается плагин в браузер, который запоминает все действия по тестированию потом информация передаётся в специальный движок (драйвер), который в состоянии эмулировать определённые типы браузеров и выполнять записанные действия.

Потом этот функционал можно интегрировать с нашими сценариями CI/CD. Например, на dev и stage окружениях функциональные тесты можно запускать после того, как новый код был доставлен на эти окружения.

Из инструментов автоматического функционального тестирования можно выделить следующие.

Инструменты функционального тестирования

Selenium

Сайт, документация.

Самый первый фреймворк для тестирования.

Есть отдельный модуль под названием **Selenium IDE**, который позволяет разрабатывать тестовые сценарии.

Он поставляется в виде дополнения к браузеру Firefox и, в целом, является наиболее эффективным способом разработки тестовых сценариев.

Дополнение содержит контекстное меню, позволяющее пользователю сначала выбрать любой элемент интерфейса на отображаемой браузером в данный момент странице, а затем выбрать команду из списка команд Selenium с параметрами, предустановленными в соответствии с выбранным элементом.

Модуль Selenium Grid позволяет запускать тесты одновременно на нескольких серверах, что уменьшает время тестирования.

Katalon

Сайт, документация.

Это более новый продукт, который работает на тех же принципах, что и Selenium. Умеет эмулировать мобильные браузеры, тестировать API. Считается преемником Selenium.

Нагрузочное тестирование (load testing)

Смысл этого тестирования заключается в том, чтобы посмотреть, как приложение будет вести себя под нагрузкой. Естественно это будет больше синтетический тест, поскольку все web-приложения работают через балансировщики нагрузки, которые держат подключения с одного ip-адреса в кэше и не инициируют новое соединение с бэкендом (алгоритм «липких» соединений).

Для таких действий тоже есть инструмент, который позволяет автоматизировать подобное тестирование.

Apache JMeter

Сайт, документация.

Тестирует практически все параметры web-приложения. Может отсылать необходимые HTTP-заголовки.

Поддерживает импорт данных для заполнения форм из csv-файла (можно забить 100 логин/пароль/email для регистрации и jmeter будет из подставлять при каждом новом соединении). Можно арендовать на время 50 виртуальных машин в разных геозонах и «бомбить» jmeter-ом со всех сторон.

Chaos Monkey

Сайт, документация.

Этот продукт можно назвать результатом обработки концепции Chaos engineering – серия экспериментов, которая должна смягчить последствия сбоев.

Очень интересная концепция. Позволяет протестировать всю инфраструктуру сервиса и надёжность архитектуры.

Суть состоит в следующем – на все сервера окружения устанавливаются модули Chaos Monkey, которые в случайном порядке начинают проводить «диверсии». По случайному алгоритму останавливают или перезагружают сервера и сервисы, удаляют данные и т.д.

> В Netflix (они её и разработали) даже есть специальные «game days» (в начале 2000-х Jesse Robbins, занимавший в Amazon должность с официальным названием Master of Disaster, создал и возглавил программу GameDay. Она была основана на его опыте пожарного. GameDay предназначалась для тестирования, обучения и подготовки различных систем Amazon, программного обеспечения и людей к воздействию потенциальных кризисных ситуаций.), когда собираются представители разных служб и запускается Chaos Monkey.

Звучит нелепо, при первом приближении, но через какое-то время начинаются проступать изъяны в архитектуре. И что самое главное – изъяны в построении процесса поддержки, мониторинг и эскалации инцидентов.

Развёртывание окружений

«Девопс» тоже хочет комфорта, он тоже хочет сделать коммит, «запушить» и получить результат. Окружений от него требуют часто, поэтому появляется такое явление **Infrastructure As Code**.

Суть явления – инфраструктура описывается в виде кода, как правило json или yaml, который потом передаётся «движку», который с ним работает. «Движок» в зависимости от параметров кода создаёт окружение – ОС, софт, «конфиги».

> Понятно, что уже есть Docker, Terraform, Amazon Cloud Formation. Это тренд и довольно подробно описан везде. Также это будет мэйнстрим на ближайшие 10 лет.

Мы бы хотели остановиться на достаточно универсальном инструменте, который очень любим использовать. Это Hashicorp Packer.

В общих чертах – язык описания инфраструктуры json. Структурно состоит из Builders – модуль, который собирает окружения, и Provisioners – устанавливает пакеты, настраивает конфиги, добавляет пользователей.

Что может собрать Builders (основное):

Alicloud ECS, Amazon EC2, Azure, CloudStack, Digital Ocean, Docker, Google Cloud, Hetzner Cloud, Hyper-V, Linode, LXC, OpenStack, Parallels, QEMU, Vagrant, Virtual Box, VMware, Proxmox (API)

Какие поддерживаются Provisioners (основное):

Ansible Local, Ansible (Remote), Chef Client, Chef Solo, PowerShell, Puppet Masterless, Puppet Server, Salt Masterless, Shell, Shell (Local), Windows Shell

Получается, что этот инструмент может ВСЕ!

Хотите собрать и выложить в виде виртуалки KVM на «железный» гипервизор – QEMU+Ansible. Хотите выложить AMI-образ в Amazon – Amazon EC2 +Salt (для разнообразия).

Можно работать с Docker. Причём контейнер собирается без Dockerfile, не надо разбираться с синтаксисом.

Самая главная концепция DevOps в этой ситуации – образ (сервер) не настраивается в режиме удалённого доступа, если что-то пошло не так. Образ создаётся заново с новыми параметрами. Целостность и унификация окружений – главная цель DevOps. Не должно быть окружений с разными системными параметрами (**configuration drift**) – ядра, версии библиотек, список софта, конфигурационных файлов.

Infrastructure as code (IaC)

Подход, при котором вся инфраструктура описывается в виде кода. Разделяется на логические блоки и «оцифровывается». Дальше процесс идёт аналогично стандартному процессу разработки – тестирование, deploy на окружения.

Что обычно входит в «кирпичи», из которых строится окружение:

- Параметры сетевых интерфейсов. IP-адреса, шлюзы, маршруты, DNS
- Список ПО, которое необходимо установить через пакетный менеджер
- Настройки необходимого ПО
- Учётные записи пользователей, с правами которых, будет работать приложение
- Учётные записи баз данных и остального
- Права доступа к директориям и файлам
- Настройки кластеров, реплик и т.д.

Благодаря тому, что в Linux все является файлом или ссылками на файл, это концепция прекрасно подходит для того, чтобы держать всё это в Git.

Что при этом необходимо от систем, которые можно использовать для работы в формате Infrastructure as code (IaC):

- **Поддержка шаблонов**. Нам надо не править конфигурационный файл для каждого сервера, подставляя параметры типа hostname или port. Хотим нужные нам параметры держать в виде переменных.
  Поддержка минимальной логики обработки задач. Например, если предыдущая задача не выполнена, то процесс останавливается. Если удовлетворяет условию №1, тогда выполняется задача №4, если условию №2, тогда выполняем задачу №3.
- **Обобщённые действия или обработчики.** Например, мы хотим вынести в отдельный процесс обновление системы и перезагрузку при успешном обновлении. И просто подключать эти процессы в сценарии при необходимости.
- **Возможность взаимодействия с системами и сервисами.** Создать пользователя, назначит права файлам, создать базу данных. Все это надо делать с помощью файлов сценариев.
- **Необходима возможность надёжно скрывать передаваемые в скрипты пароли и другие учётные данные.**
- **Нужна возможность исследования хостов, которые будут настраиваться с помощью сценариев.** Потом на основе этих данных использовать нужную нам логику. Например, если операционная система хоста Debian, то в скриптах установки ПО будет использоваться команда apt-get, если CentOS – тогда rpm или yum.

- **Возможность централизованной работы с наборами сценариев.** Единый сервер, который будет все запускать.

В данный момент основными фаворитами в данном классе ПО, которые набирают популярность являются **Ansible** и **Salt**.

Ansible

Сайт, документация, готовые роли и плейбуки.

Безусловный фаворит на рынке. Самый низкий порог вхождения. Написан на Python не требует агентов на управляемых серверах, только установленный Python.

С Windows работает через службу WinRM (Windows Remote Shell).

Все сценарии описываются на yaml. Для создания шаблонов используется шаблонизатор Jinja2.

Поддерживает большой список модулей управления – начиная от баз данных и заканчивая некоторыми сетевыми устройствами. Полный список модулей.

Позволяет запускать как одиночные команды, так и сценарии, которые в терминологии Ansible называются playbook.

Состоит из следующих компонентов:

- **Ad-hoc command** – запуск одномоментной команды на удалённом сервере или в «сыром виде» или с помощью модулей.
  Inventory – инвентаризация управляемых хостов. Сбор информации о хосте с последующим её использованием в сценариях. Начиная от MAC-адреса и заканчивая списком установленного ПО. Также используется для описания групп хостов, которые будут управляться.
  Vars – переменные. Используются переменные как одного хоста (host_vars), так и группы хостов (group_vars)
  Template – шаблон. Используется синтаксис шаблонизатора Jinja2, в который можно вставлять переменные. Как пример, можно привести стандартный файл виртуального хоста в Apache, у которого параметры *ServerName*, *DocumentRoot*, *ServerAlias* будут указаны в виде переменных. Значения переменных будут указаны в свойствах каждого конкретного хоста, который надо будет настроить.
  Module – модуль. Написанный, как правило на Python интерфейс управления к службе или сервису. Используется в сценариях.
  Playbook – набор команд, которые выполняются по очереди. Описанный на yaml.
  Role – предустановленный набор переменных, playbook, которые решают определённою задачу. Например, установить сервер баз данных MongoDB. Роли можно вызывать из других Playbook, используя специальный параметр roles.
  Vault – хранилище. Зашифрованное хранилище конфиденциальных данных, которые используются в playbook.

Благодаря широкой поддержке и популярности уже есть целый портал с готовыми ролями на все случаи жизни. Но рекомендуем сильно не увлекаться, поскольку роли как правило, пишутся под свои нужды и можно потратить время на изучение и изменение «под себя». В итоге окажется полезней и надёжней написать роль самому. А в будущем быстрее и надёжней.

Для централизованной работы с playbook и хостами через web-интерфейс, Red Hat (владелец Ansible) предлагает коммерческий продукт Ansible Tower, также есть бесплатная версия – AWX.

Salt

Сайт, документация, готовые роли.

Ещё один продукт для оркестрации.

Тоже написан на Python. Для описания задач использует yaml.

Шаблоны описываются с помощью Jinja2. Поддерживает все основные архитектурные возможности Ansible – шаблоны, переменные, playbook, только называются по-другому. Имеет клиент-серверную архитектуру.

Состоит из следующих компонентов:

- **Grains** – инвентаризация хостов, аналог Inventory у Ansible
- **Minions** – «помощники», агенты, которые запускаются на управляемых хостах и связываются с сервером Salt
- **Salt Master** –сервер Salt, к которому подключаются Minions.
- **States** –состояния, к которому агенты (Minions) должны привести управляемые хосты. Своеобразный аналог Playbook в Ansible
- **Pillar** – общее глобальное состояние (States) для всех хостов. Аналог Roles в Ansible

Продукт несколько своеобразный, но очень мощный. В некоторых случаях он предлагает намного более мощный чем в Ansible интерфейс управления модулями.

Централизованный сервер управления с web-интерфейсом доступен только в платной версии.

Оба этих продукта имеют отличную документацию и обязательны к изучению на самом глубоком уровне, поскольку это один из основных инструментов DevOps.

Мониторинг

Тема мониторинга – огромный пласт материала для изучения и тестов.

Систем очень много, как и различных типов данных, которые необходимы для мониторинга и анализа. Также это все очень тесно переплетено с логированием.

Для начала надо определиться, что будем мониторить:

- Параметры мощностей, на которых работает сервис или приложение. Загрузка CPU, память, место на диске. Стандартные «админские» параметры.
- Все, что связано с пропускной способностью внешнего периметра. Скорость на входящем интерфейсе, соотношение входящего/исходящего трафика.
- Доступность самого сервиса. Автоматические сценарии доступа ко всем endpoint сервиса, API и т.д. Коды ошибок, скорость обработки запроса. TTFB – время от нажатия ссылки в браузере до получения первого байта от сервера.
- Метрики сервисов и приложения. Память, CPU, I/O дисков потребляемая базами данных, запросами, кэшами, быстродействие самого приложения, служебные процессы.
- Метрики для бизнеса. Количество регистраций, скорость прорисовки страницы, скорость самой регистрации – от момента, когда пользователь зашёл на страницу до момента, когда данные попали в базу.
- Метрики для отдела безопасности сервиса. Неудачные попытки входа в кабинет, вал запросов на API сервиса, сканирование портов и т.д.

> Сейчас мониторинг и особенно управление метриками это отдельная, большая дисциплина, которой занимаются, как правило, специальные сотрудники – Monitoring Engineer.

Вот прекрасная статья на тему методологий по которым рассчитываются ключевые метрики.

Перед внедрением системы мониторинга необходимо тщательно подготовиться:

1. Проговорить, обсудить, утвердить метрики, которые необходимы бизнесу. Это очень огромный кусок работы на самом деле. Загрузка процессора бизнес не волнует. Бизнес волнуют прибыли. Поэтому мониторинг должен быть предназначен в первую очередь для бизнеса. Бизнесу важно знать, что происходит, когда он даёт рекламу – мы должны рассказать, как растут регистрации после того как пошла реклама, выдержит ли сервис одномоментный приток новых пользовате-

лей. Сколько было регистраций на сервисе. Что в инфраструктуре стало работать хуже после наплыва посетителей на сайт. Как растёт загрузка базы данных после того, как пользователь делает выписку по счету, например. Эти параметры, которые мы передаём, и есть значительная часть концепции «time to market», ради которой DevOps и привлекается.
2. Мы должны знать базовые значения. Какая у нас загрузка инфраструктуры при линейной, спокойной работе. Чтобы было с чем сравнивать и кричать «караул», когда идёт отклонение от базовых значений.
3. Разработчики должны включать в код endpoints для бизнес-метрик.

Требования к системе мониторинга:

1. Работа под большой нагрузкой. Очень важна архитектура баз данных, промежуточных хранилищ, сетевой подсистемы. Данных может быть очень много из разных источников.
2. Агенты для всех операционных систем. Понятно, что можно написать зоопарк обвязок к системным утилитам (спойлер – в некоторых случаях придётся именно так и поступать) и радоваться, что используется «безагентская» система мониторинга «под себя». Но самый надёжный способ работать через агентов.
3. Поддержка SNMP
4. Настраиваемые дашборды и экраны
5. Построение карты инфраструктуры. Огромная помощь для отдела эксплуатации видеть мониторинг в реальном времени в разрезе общей картины в виде сетевой схемы. «Проблемные» узлы выделяются, например, красным светом.
6. Возможность гибко автоматически «чистить» хранилище данных. Система, например, удаляет данные старше 3-х месяцев.
7. Возможность расширять функционал плагинами и шаблонами.
8. Возможность совершать программируемые математические и статистические операции. Например, прогнозирование трендов.
9. Наличие API
10. Защищённое соединение между сервером и агентами
11. WEB-мониторинг. Отправка запросов на web-сервер, мониторинг кодов ответа сервера.
12. Автоматическое выполнение действий по триггерам. Например, перезапустить сервис на наблюдаемом instance, если сервис не отвечает или выдаёт ошибку
13. Отправка алертов на почту, в мессенджеры, push уведомления
14. Мониторинг производительности приложения (APM)

Инструменты мониторинга

Zabbix

Сайт, документация, шаблоны для мониторинга, обзор решений и возможностей мониторинга в разрезе технологий и объектов (рекомендуется к изучению).

Прекрасная система, которая подходит для быстрого внедрения мониторинга.

Мониторинг основных «админских» параметров – диск, трафик, скорость интерфейсов, загрузка процессора. Графики, дашборды и т.д. Алерты, плюс хорошая возможность совершать действия на хостах при срабатывании определённого+ триггера – перезапуск сервиса и т. д.

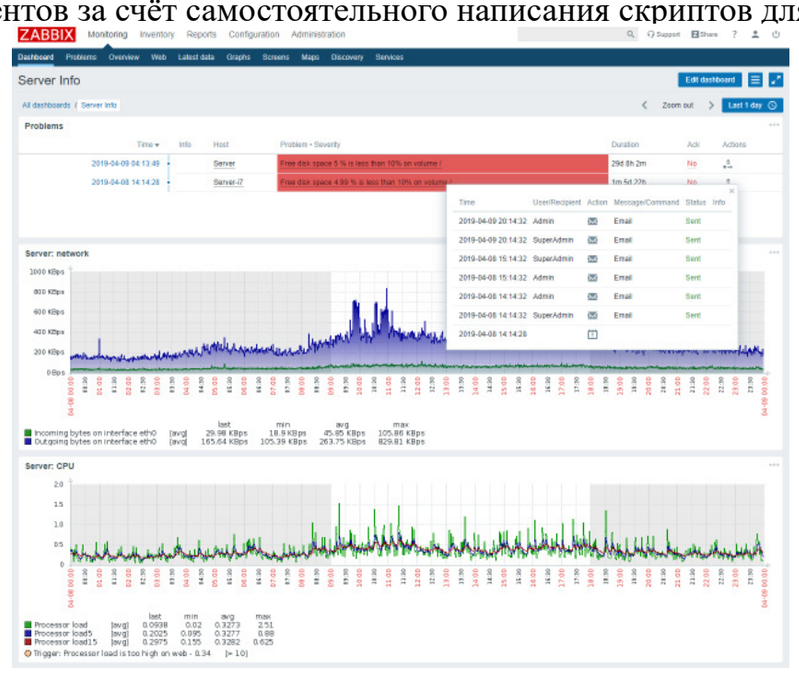
Быстро устанавливается, быстро настраивается. Работает через агентов, поддерживаются MacOS, Windows, Linux, FreeBSD, OpenBSD. Есть мониторинг по SNMP, IPMI.

Отдельным пунктом стоит упомянуть про мониторинг web-приложения, триггеры на коды возвратов web-сервера.

Можно нарисовать сеть сервиса (не сам). Если есть потребность в добавлении боль-

шого количества узлов для мониторинга, можно использовать функционал обнаружения (модуль Discovery) – задать подсеть с агентами и Zabbix сам будет добавлять хосты в рамках заданного интервала.

Есть хорошее сообщество, много документации. Можно расширять функционал агентов за счёт самостоятельного написания скриптов для мониторинга нужных параметров.



В качестве базы данных MySQL или PostgreSQL. Фронтенд написан на PHP. С версии 4.4 произошёл прорыв – переписаны агенты, есть официальная поддержка TimescaleDB – база данных с открытым исходным кодом для хранения временных рядов, основанная на PostgreSQL.

Сильно «вырос» с выходом 5-й версии. Появилось много шаблонов монитринга «из коробки», стало возможно хранить ключи доступа в сторонних хранилищах типа Hashicorp Vault, появидся мониторинг IoT устройств. Даже есть встроенная интеграция с Jira и Slack.

В общем – заслуженный продукт с долгой историей.

Prometheus

Сайт, список официальных экспортёров, документация.

Список библиотек для написания экспортёров для разных языков программирования.

Второй общепризнанный лидер – Prometheus. Вернее, скоро он будет абсолютным лидером в системах мониторинга.

По сути это просто база, которая аккумулирует данные из экспортёров (агентах) на клиентах. Есть огромное количество экспортёров уже готовых и есть прекрасные возможности их писать самому и подключать к коду. Это намного более универсальный инструмент чем Zabbix, но и более сложный и к нему надо подходит с продуманными и готовыми требованиями.

Для хранения и обработки данных мониторинга используется специальная time-series база InfluxDB.

Данные с экспортёров передаются по HTTP-протоколу.

Для оповещений используется отдельный продукт Alertmanager.

Запросы к базе выполняются на специальном языке PromQL (Prometheus Query Language)

Визуализация

Prometheus это просто очень умная база со своим языком запросов. Поэтому надо визуализировать данные. Для этого есть прекрасный продукт – Grafana.

Grafana

Сайт, список плагинов, список готовых дашбордов.
Признанный лидер визуализации данных.
Целый коктейль технологий.

Позволяет визуализировать данные из нескольких источников:

- Graphite
- Prometheus InfluxDB
- Elasticsearch
- Google Stackdriver
- AWS Cloudwatch
- Azure Monitor
- Loki
- MySQL
- PostgreSQL
- Microsoft SQL Server
- OpenTSDB (например, InfluxDB)
- MixedData на основе разных запросов к разным базам данны

> Может использовать данные для визуализации из Zabbix

В итоге имеем очень мощный и гибкий конструктор для визуализации данных из разных источников.

Мониторинг производительности приложения (APM)

Огромная тема для изучения, исследований и тестирования необходимых решений.

Описанные выше инструменты созданы для мониторинга определённого набора сервисов и приложений, которые имеют в основной своей массе открытый исходный код, имеют многолетнюю историю, известны на рынке и являются стандартами. Они изучены инженерами, которые создавали и разрабатывали системы мониторинга.

Как быть, когда надо мониторить разрабатываемое приложение? Не просто один модуль, а сложный сервис, у которого много зависимостей, ролей, функций. И знают об этом сервисе только разработчики. Можно даже сказать больше – по сути, для получения прибыли и развития продукта, нам надо мониторить только наше приложение.

С вышеперечисленными средствами это можно делать только по каким-то косвен-

ным признакам: сетевой трафик, нагрузка на базу, много 500-х ошибок от web-сервера. Получается, что мы узнаем, что что-то случилось, когда мы ощущаем какой-то «дискомфорт».

Для решения таких задач есть специальный класс продуктов – APM (**Application Performance Monitoring**). Суть проста, а реализация достаточно сложная. Для каждого языка программирования есть библиотеки, которые надо использовать в коде для создания метрик, которые интересуют. Метрики во время работы отправляются на сервер мониторинга. Какие метрики можно включать в мониторинг – дело вкуса, потребностей и т. д., поскольку писать их все равно будет команда и давать задание будет «бизнес».

Как пример можно привести стандартный web-сервис:

Фронтенд:

- Скорость рендеринга страницы на фронтенде
- Скорость кеширования сессий, количество сессий
- Ошибки на фронтенде
- Скорость соединения с CDN если используется
- Скорость и количество запросов на бэкенд в пределах одной сессии пользователя

Бэкенд:

- Скорость доступа к базе, медленные запросы
- Скорость и количество запросов к подрядчикам (типа платёжных систем)
- Количество ошибок на бэкенде в разрезе запросов

Полет фантазии и потребностей неограничен. Повторимся ещё раз – тема для изучения огромная и займёт целую книгу.

Инструменты APM

Сразу вынуждены огорчить бесплатных продуктов в этой области мало.

Создание APM-системы достаточно трудоёмкий и дорогой процесс.

Как обычно, опишем необходимые требования:

1. Библиотеки и документация для широкого спектра языков программирования. Разработчик должен получить мощный и хорошо документированный инструмент для сбора необходимых метрик
2. Построение динамических схем и карт. С учётом массового перехода на микросервисы и системы оркестрации необходимо видеть динамику контейнеров – сколько создалось, сколько удалилось, нагрузка на pod-ы
3. Визуализация – графики, диаграммы и т. д.
4. Алертинг – стандартный набор из sms, push-уведомлений, email, мессенджеров

В общих чертах нам надо увидеть, что происходило со всем нашим сервисом, когда был сбой в каком-то модуле.

NewRelic

Сайт, полный список возможностей APM.

Лидер рынка.

Возможности:

- Мониторинг приложений. Время отклика, пропускная способность и частота ошибок на фронтенде. Список наиболее трудоемких веб-транзакций предоставляет агрегированные сведения о медленных транзакциях, которые происходят в течение указанного временного промежутка, а также общее время ответа для каждой из них.
- Мониторинг баз данных. Обеспечивает подробный обзор производительности базы данных, указывая на критические ошибки, замедляющие работу приложения.
- Доступность и мониторинг ошибок. Оповещения позволяют устанавливать собственные предупреждения и критические пороги, чтобы можно было быстро находить и решать проблемы.
- Внешний Pinger, который позволяет быть уверенным, что клиенты попадают на

сайт. Pinger проверяет доступность приложений, регулярно отправляя им запросы, регистрируя ошибки и отправляя предупреждения о простоях, когда приложение не работает.

> В пакете услуг Standard дается доступ для одного экаунта и 100 GB/месяц бесплатной обработки данных.

Datadog

Сайт, возможности, с чем интегрируется.

Поддерживаемые фреймворки – Laravel, ASP. NET MVC, Django, Ruby on Rails, Gin, Spring

Atatus

Сайт, возможности, с чем интегрируется.

> Из бесплатных APM можно отметить Pinpoint. Позволяет подключать функции APM для проектов на PHP.

Sentry

Сайт, документация, список поддерживаемых языков и платформ.

Open source система для мониторинга ошибок приложений. Имеет свой SDK, который надо использовать при разработке. Все ошибки фиксируются на выделенном сервере, к которому есть доступ через web-интерфейс.

> Есть интересная возможность встроить модуль «User Feedback» в свой продукт, который добавляет возможность сбора дополнительной обратной связи от пользователя при обнаружении ошибки.

Jaeger

Сайт, документация.

Open source трассировщик для приложений на основе микро сервисов. Разработан Uber Technologies.

Резюме

Качественный мониторинг процесс не одно спринта. Это постоянная работа по улучшению, развитию, изучению. В идеале мониторинг должен быть первоочередной задачей отдельного специалиста, на котором будут следующие обязанности:

- Постоянная обратная связь с бизнесом по развитию бизнес-метрик
- Постоянная связь с маркетингом, чтобы знать о мероприятиях, которые могут привести к росту нагрузки на сервис
- Помощь разработчикам

Логирование

Мы специально не объединяли в одном разделе мониторинг и логирование – получился бы большой объем материала. Это вроде схожие по смыслу про-

цессы, но дьявол кроется в деталях. Мониторинг – это больше к повседневной и оперативной работе.

Логирование относится больше к «разбору полётов». Хотя инструментарий может местами поддерживать оба этих процесса.

Понятно, что вопрос важности ведения и хранения журналов системы даже не обсуждается.

Рассмотрим вопросы:

- Где хранить
- Для кого хранить
- Сколько хранить
- Как обрабатывать

Стандартный вариант – настраиваем syslog сервер, настраиваем все наши сервисы во всей инфраструктуре писать log-и на удалённый сервер. В итоге получим все log-и в одном месте. Потом берём в руки awk, grep и получаем все, что нам надо от log-ов.

> В реальном мире, при работе команды разработчиков, появляется несколько вещей, которые рушат всю идеальную картину такой схемы.
>
> Awk, grep знают не все на приемлемом уровне, чтобы использовать их продуктивно; мгновенно вычисляется товарищ, который знает awk, grep и превращается коллегами в инструмент по парсингу логов. Особенно отчаянные начинают писать связки Perl+apache, чтобы их не трогали.

Огромный плюс хранения log-ов на удалённом сервере – закрытие одной из хакерских лазеек по удалению или подмене файлов журналов для сокрытия следов. С другой стороны – если взломают сервер хранения логов… то ой.

Где хранить log-и

Сразу можно сказать, что мощности для сервера нужны более чем солидные. Помимо хранения будет ещё и обработка, индексация, поиск. Не забываем про подключение с хорошей пропускной способностью. Это относится к любому агрегатору и парсеру log-ов.

Естественно отказоустойчивое хранилище большого объёма – в зависимости от специфики сервиса, возникнет необходимость хранить «сырые» log-и годами. Хорошая практика – шифровать файловую систему на всех разделах сервера-агрегатора. При росте данных стоит предусмотреть возможности кластеризации системы.

Сколько хранить log-и

Стоит различать «сырые» log-и и обработанные. «Сырые» log-и могут понадобится совершенно внезапно и в разных ситуациях: запрос по судебным решениям (в зависимости от страны), разбор взломов, проведение исследований по деградации сервиса и т.д. Даже при самых оптимистичных ситуациях мы рекомендуем срок хранения в 12 месяцев.

Обработанные log-и хранятся в базе и их можно получить снова в любой момент времени, «пропарсив» заново «сырые».

Для кого хранить log-и

Когда ты администратор или DevOps про доступ не задумываешься. Когда понимаешь, что смотреть могут разные люди, то начинаются уже вопросы безопасности и т.д.

Хорошее правило – рассматривать доступ к централизованному хранилищу log-ов в разрезе общей информационной безопасности: права доступа, группы, центральная авторизация через LDAP или Active Directory.

В идеале фронтенд видит log-и web-сервера, бэкенд log-и баз данных и своих приложений, «безопасники» log-и авторизаций и т.д.

Как обрабатывать log-и

Иногда случаются ситуации, когда в log-и попадает конфиденциальная информация

– пароли, номера банковских карточек, IP-адреса и т. д. То, что не для широкой публики.

Снова напомним о «сырых» и обработанных логах. Можно скрывать нужные нам слова и строки при обработке, или сделать так, чтобы они вообще туда не попадали.

С обработкой все понятно – специальный фильтр или regexp и все спрятано.

Другая ситуация с сокрытием данных в «сырых» файлах. Все это необходимо проектировать и внедрять на уровне сервиса или приложения. Например, в Percona MongoDB есть встроенный параметр:

```
security:
  redactClientLogData: true
```

который заставляет сервер скрывать конфиденциальные данные при записи log-ов.

Поэтому необходимо тщательно изучить документацию используемых решений при проектировании сервиса или приложения.

Если разработчики хотят включать логирование в свои приложения, то также необходимо предусмотреть такой механизм при проектировании.

Требования к инструментарию для агрегации log-ов

Минимальный список интересующих возможностей ПО аккумуляции и обработки log-ов:

1. Поддержка всех возможных форматов и протоколов журналирования
2. Широкие возможности по обработке log-ов, анализ данных log-ов, комбинирование из разных источников
3. Поиск и индексация
4. Визуализация
5. Сигнализация по определённым данным или событиям в log-ах
6. Возможность расширения функционала через плагины или через API
7. Шифрование передачи log-ов
8. Ролевая модель доступа

Опишем эти требования более подробно.

Поддержка всех возможных форматов и протоколов журналирования

В большинстве случаев log-файл представляет собой обычный текстовый файл. Со временем программное обеспечение начало усложнятся и в некоторых случаях текстовый файл перестал удовлетворять потребности разработчиков. Началась эволюция форматов, в которых хранятся log-и.

Самые распространённые в данный момент:

- Syslog
- GELF(Graylog Extended Format logging)
- AWS - AWS Logs, FlowLogs, CloudTrail
- Beats/Logstash
- CEF (Common Event Format)
- JSON
- Netflow
- Plain/Raw Text
- Apache Log4j

Каждый формат имеет свои особенности и для него требуется своя логика «парсинга» и обработки.

Самый простой способ сбора log-ов по UDP-протоколу, он достаточно быстрый, простой, и данные передаются быстрее. В некоторых случаях может понадобится HTTP API для сбора log-ов.

Широкие возможности по обработке log-ов, анализ данных log-ов, комбинирование из разных источников

Например, мы хотим собирать все log-и от MongoDB на UDP port 1067, у нас 57 MongoDB серверов по всем нашим проектам и окружениям. Соответственно нам будет необходимо различать эти сервера по каким-то параметрам: hostname, IP-адрес или тэг.

Следующий этап – разумно объединить обработку логических событий, которые нас интересуют:

1. Список всех неправильных логинов со всех серверов, по интересующим нас сервисам – безопасники должны видеть с каких адресов вводят неправильные логины/пароли для доступа к SSH, http (различные web-интерфейсы, кабинеты пользователей)
2. 400-500 ошибки с web-серверов для фронтэнда

Построить графики, которые показывают динамику предыдущих пунктов.

Вроде несколько простых действий, а работы системе придётся проделать достаточно много.

Поиск и индексация

При таком объёме данных поиск приобретает огромное значение. Необходимо хранилище с такой архитектурой, которая позволяет работать с временными метками – это основное и главное свойство любой системы логирования.

Для таких случаев на рынке начали развиваться так называемые time-series databases – базы данных временных рядов.

Временной ряд – это статистика серии наблюдений за одним и тем же явлением, параметром какого-либо процесса, на протяжении некоторого времени.

Каждому результату наблюдения (измерению) соответствует время, когда это наблюдение было сделано. Таким образом, при анализе временных рядов учитываются не только базовые статистические закономерности, но и взаимосвязь измерений со временем.

Сейчас есть несколько продуктов из этой серии:

- InfluxDB
- Timescale DB
- OpenTSDB
- Graphite (система сбора данных со своим решением time-series database)
- Victoria Metrix

Соответственно необходима система, которая позволит работать с одним из этих бэкендов.

Визуализация

Массив данных очень трудно воспринимается в виде таблиц или вывода в json. Для полноценного анализа и принятия решений нужны графические средства – диаграммы и графики разных видов.

Особенно полезными будут географическая визуализация.

> Для примера – раскраска стран по интенсивности перебора пароля к сервису. Если пойти дальше, то можно сделать такую же раскраску по соотношению «попытка перебора пароля» / «регистрация нового пользователя в сервисе». Самые «насыщенные» на такой карте регионы можно просто отключать по GeoIP.

Сигнализация по событиям в log-ах

Постоянно в log-и никто не смотрит. Да и не должны. Естественно нужна возможность сигнализации системы по каким-то параметрам, которые мы им зададим.

Например, когда в разрабатываемом приложении встречается что-либо вроде «fatal error», то отправляется письмо на support@company.com, с текстом записи из log-а.

Естественно, чем шире возможности коммуникаций, тем лучше – электронная почта, мессенджеры, смс.

Также необходимо чтобы была возможность настроек отправки алертов по разным рабочим группам и пользователям. Огромным плюсом будет автоматическое создание отчётов по настраиваемым параметрам.

Возможность расширения функционала через плагины или через API

Внедрение системы централизованного логирования это трудозатратный и дорогостоящий процесс.

Естественно всегда будет чего-то не хватать в штатном функционале.

Огромное значение при выборе системы играет сообщество пользователей, которое начинает предлагать улучшения и плагины.

Защита передачи log-ов

Разделить можно на несколько частей:

- Защита соединения между сервером агрегации и клиентами
- Обсфукация (затенение) данных в самом передаваемом log-е.
  Первый вариант можно реализовать самостоятельно – построить VPN-тоннели между сервером и клиентами. Это немного замедлит передачу, но так получится один из уровней защиты.

Второй вариант более ресурсозатратный. Мы выше писали, что «затенение» реализуется на уровне приложения, которое пишет log-и. Как вариант, можно использовать TLS-шифрование траффика, который передаётся через syslog, например.

Ролевая модель доступа

Агрегатор логов это один из самых главных в плане безопасности сервисов в компании.

Он должен быть максимально защищён:

- Доступ сотрудников только по VPN – но одного сетевого порта не должно быть снаружи.
- Данные от клиентов передаются тоже только по VPN. Это очень трудозатратный процесс – создать общую VPN сеть между наблюдаемыми клиентами и агрегатором логов, но это того стоит.
- Все диски сервера, пусть даже это instances на AWS, должны быть зашифрованы. Все бэкапы должны быть зашифрованы. Диски бэкапного сервера должны быть зашифрованы.
- Также необходима ролевая модель доступа к обработанной и «сырой» информации. Выше мы писали про разные группы типа безопасности и фронтэнда со своими данным. Как один из вариантов – разнообразные подрядчики сервиса должны иметь доступ к log-ам, которые связаны с их сервисами.
- Для удобства администрирования и совместимости с остальными продуктами информационной системы разработки нужна поддержка централизованных систем управления пользователями – OpenLDAP, Active Directory.

Продукты для агрегации и обработки log-ов

Graylog

Сайт, документация, плагины.

Самый целостный в плане вышеперечисленных требований продукт.

Поддерживает авторизацию через OpenLDAP, Active Directory, также есть возможность двухфакторной аутентификации.

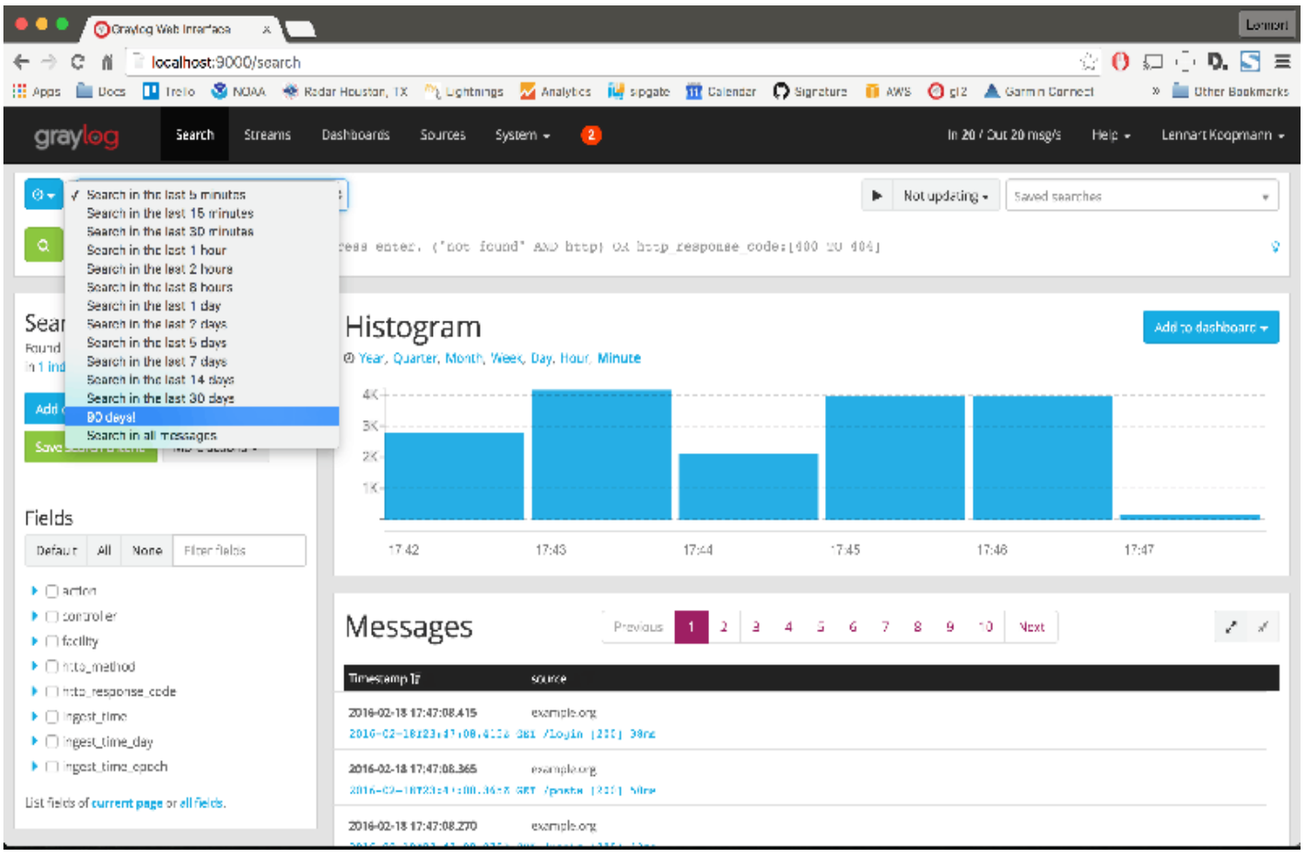
Структурно, на уровне приложения, состоит из нескольких абстракций:

- **Inputs** – входной поток для сообщений с клиентов. Создаётся в виде «тип логирования»: «протокол»: «порт»
- **Outputs** – пересылка принятые сообщения другим системам по мере их обработки
- **Event Notifications** – вызывается, когда срабатывает определённый триггер, и отправляется «алерт»
- **Processors** – трансформирует или отбрасывает входящие сообщения из разных

источников

- **Decorators** – могут быть использованы для преобразования поля сообщения во время отображения
- **Authentication Realms** – имплементируют различные механизмы аутентификации

Архитектурно состоит из MongoDB и Elasticsearch, написан на Java



ELK Stack

Сайт, документация.

Стандарт де-факто в отрасли. Является акронимом от трёх проектов: **Elasticsearch, Logstash, Kibana**.

Но есть ещё один компонент – **Beat**, который почему-то не указан.

Logstash – это утилита для сборки, фильтрации и последующего перенаправления в конечное хранилище данных. Вход-фильтр-выход. Плагины для Logstash.

Elasticsearch – это решение для полнотекстового поиска и индексации.

Beat – агенты, которые устанавливаются на клиентах для сбора log-ов. Может направлять данные непосредственно в Elasticsearch или в Logstash.

Поддерживаются следующие типы передаваемых данных:

- Audit data – Auditbeat. Проверка и мониторинг целостности файлов.
- Log files – Filebeat. Агрегирование логов
- Cloud data – Functionbeat. Агрегирование логов с облачных провайдеров AWS
- Availability – Heartbeat. Мониторинг доступности web-сервисов. Отсылает запрос, анализирует полученные данные в Elasticsearch, Kibana визуализирует.
- Metrics – Metricbeat. Снятие стандартных показателей с «железа» реального и виртуального. Загрузка процессора, объем диска, расход памяти и т.д.
- Network traffic – Packetbeat. Мониторинг сетевого трафика в зависимости от протокола. В том числе и высокоуровневые типа HTTP
- Windows event logs – Winlogbeat. Сбор данных с Windows event logs

Kibana.

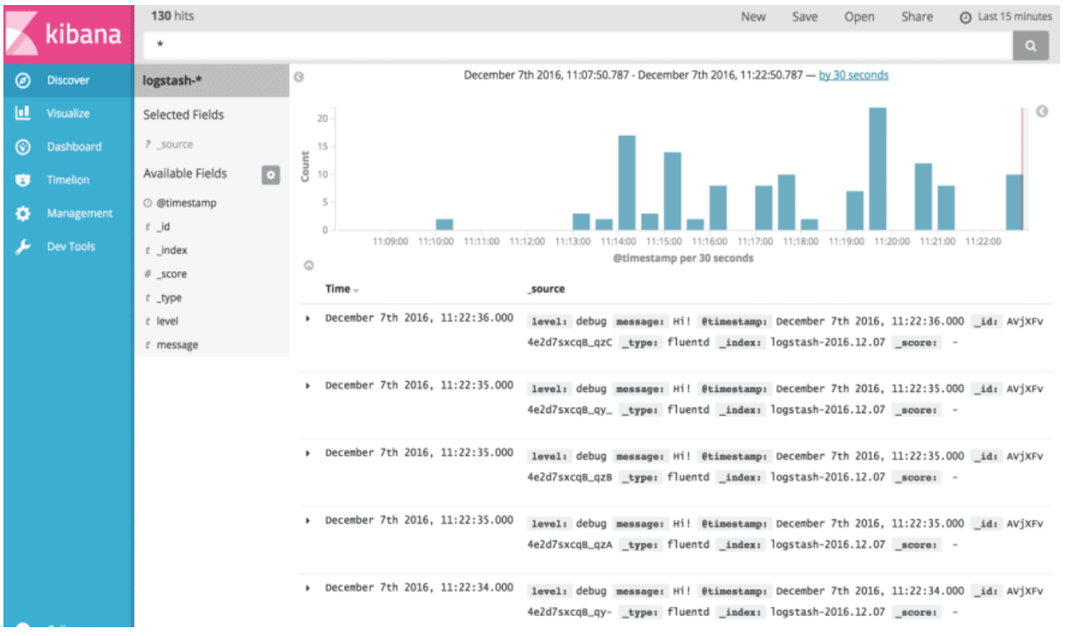
Документация, плагины.

Решение по визуализации на Angular.js. Позволяет строить графики, диаграммы, «дашборды».

Поддерживаются следующие типы визуализации:

- Базовые схемы – графики, площади, гистограммы, карты температур
- Данные – таблица данных, метрика

- Карты – карта координат (сопоставляет данные агрегации и географическое месторасположение), карта регионов (тематические карты, где интенсивность цвета формы соответствует метрическому значению)
- Временные ряды – Timelion (вычисляет и объединяет данные из нескольких наборов данных временных рядов), Time Series Visual Builder (визуальный конструктор временных рядов)



По сути ELK огромный, мощный конструктор, который ограничен только фантазией команды. В чём-то может заменить такие системы мониторинга как Zabbix, Prometheus.

Естественно, если надо получить просто агрегатор log-ов с простейшими параметрами выборки и анализа, то это решение несколько избыточное. Но если надо вести «тонкий» мониторинг и исследования, то это единственный доступный всем на рынке инструмент.

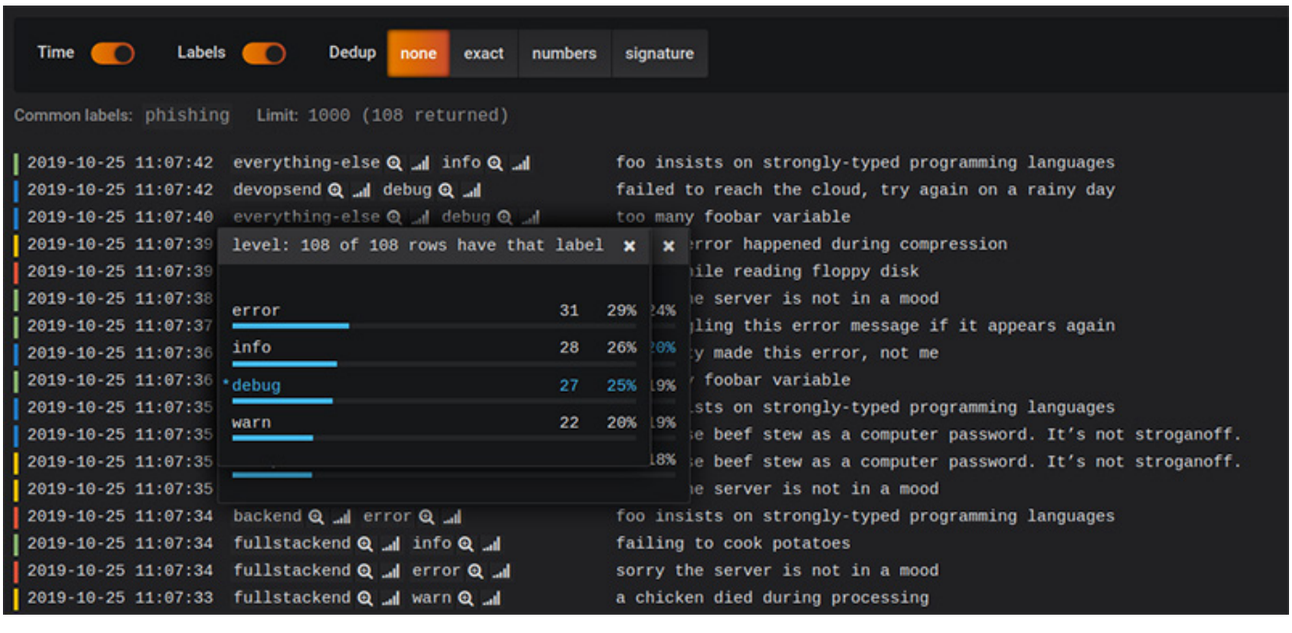
Loki

Сайт, документация.

Достаточно «простой», но симпатичный и функциональный продукт. Быстро развивается благодаря своей простоте и полной интеграции с такими «монстрами» мониторинга как связка Prometheus+Grafana.

Разработчики решили максимально идти в ногу со временем и заявили поддержку логирования Kubernetes «из коробки».

Достаточно удобно выбрать в Grafana всплеск кривой на графике и получить соответствующие log-и.



Документация

Бич современного ИТ – плохая, не полная документация или её полное отсутствие. Все меняется очень быстро и разработчики, администраторы, «девопсы» просто не успевают писать и обновлять. Задача – максимально облегчить разработчикам написание документации.

Для этих целей есть концепция **Documentation as Code(DocOps)**, в которой есть такие основополагающие принципы:

- Совместная разработка. Требуется быстрая среда для совместной работы, связанная с процессом непрерывного обновления.
- Агрегация из нескольких источников. Вместо того, чтобы хранить контент в разных местах и в отдельных хранилищах, все объединяется в одном месте.
- Интеграция с клиентами и потребителями документации: включает в себя механизм предоставления клиентам обратной связи, связанной с контентом. Это позволяет быстро обновлять и развёртывать изменения.

Какие плюсы внедрения процессов DocOps:

- **Значительно уменьшается время вхождения нового разработчика в проект.** «Смотри код, что не понятно спросишь или меня, или Петю, но Петю только по бэкенду и то, если вообще не можешь разобраться». Такого подхода уже не будет. Портал со схемой сервиса, документацией по API и т.д. избавляет новичков от «спроси Петю». «Фактор автобуса» начинает постепенно уменьшаться.
- **Уменьшается количество обращений в техподдержку.** Вернее, упорядочивается поток обращений. Пользователи не читают документацию по своему желанию. Они будут её читать если их туда направит логика скрипта процесса техподдержки или чат-бот на сайте. Написанная доступным языком справочная система помогает в большинстве случаев.
- **Растёт ценность компании.** Покупают ведь не только продукт, пользователей. Покупают и команду, и процесс разработки. Наличие хорошей документации позволит новому владельцу быстрее интегрировать другие свои продукты или сервисы, расширить отдел разработки.

Для построения DocOps необходимо ответить на ключевые вопросы.

Что будем документировать?

Отвечаем на вопросы – «как работает команда?», «как работает сервис?», «от чего зависит?», «где и на чём работает сервис?», «что нельзя трогать руками?», «у кого спросить?».

В итоге получаем:

- Документация по общей схеме сервиса или приложения. Что с чем связано. Роли. Желательно в виде схемы или диаграмм.
- Документация по инфраструктуре сервиса. Хостинг, архитектура виртуализации, список ПО, версии и т.д.
- Сторонние сервисы, подрядчики. Кто за что отвечает в нашем сервисе или приложении.
- Проблематика или технический долг. Что именно сейчас плохо, и что может плохого произойти.
- Как работает команда. Системы ветвления. Правила оформления коммитов, pull requests. Кто главный по каким направлениям и модулям. Кто принимает решения и по каким вопросам.

Как будем документировать?

Формулируем следующие требования:

- Способ должен быть максимально доступным для разработчика, начиная от редактирования и заканчивая инструментом
- Способ должен быть простым
- Необходимо укладываться в парадигму и в процессы разработки
- Минимум затрат на переносимость в другие форматы

Итог:

1. Документация должна писаться в текстовом формате и текстовом редакторе, поддерживать минимальные возможности редактирования (ссылки, картинки, стили оформления)
2. На входе мы должны получать несколько распространённых форматов
3. Документация должна храниться в системе контроля версий
4. Документация должна проходить этап тестирования
5. Документация должна автоматически собираться и выкладываться в логике Continuous Delivery

Получаем следующий алгоритм работы DocOps, самый простой шаблон:

1. Документация разрабатывается как простой текстовый файл с языком разметки в IDE или любимом редакторе разработчика
2. Файл «пушится» в репозитории
3. В зависимости от сценария сборки происходит экспорт конфигурации в:
   html со структурой и ссылками, который выкладывается на web-сервер
   docx, pdf, chm
4. Происходит тестирование документации, «прокликивание», вычитка
5. Исправления ошибок и изменения производятся в исходном файле и оформляются в виде pull requests

Инструментарий для DocOps.

Языки разметки.

- Markdown
- Asciidoctor
- reStructuredText

Все поддерживают ссылки, рисунки, стили оформления, таблицы.

Markdown самый популярный в данный момент. Под все языки разметки есть плагины для разных IDE и редакторов кода.

Конверторы

Задача конвертора из языка разметки получить на выходе другие форматы. Это поможет делать различные off-line руководства пользователя, которые могут быть скачаны с сайта или высланы по электронной почте.

Самый распространённый конвертор форматов файлов Pandoc.

Выходные форматы:

- (X)HTML 4
- HTML5
- Microsoft Word docx
- OpenOffice/LibreOffice ODT
- OpenDocument XML
- Microsoft PowerPoint
- EPUB
- FictionBook2
- PDF через дополнительные конверты типа pdflatex

Также очень полезная возможность конвертации docx в форматы md:

pandoc -f docx --extract-media images -t markdown -o document.md document.docx

Его возможности почти полностью покрывают необходимый функционал, который мы описывали.

Сервера хранения документации

Требуются минимальные возможности:

- Отображение html
- Поиск на сайте
- Понятная структура навигации

Какими инструментами это реализовать?

Самый простой способ: связка обычный web-server+генератор статических сайтов + конвертор.

Генератор статических сайтов сейчас достаточно модное направление.

Это просто – git push на входе и статический html на выходе.

Это быстро – нет запросов к базе данных для прорисовки страниц, не надо устанавливать, настраивать php и т.д.

Это экономно – поскольку задача сервера показать группу html-страниц и все, поэтому можно арендовать самые дешёвые мощности.

Как работает генератор статических сайтов. Это скрипт, которому в качестве аргумента дают папку с файлами в markdown (или другими языками разсметки) и на выходе получается папка с html с ссылками, картинками и т.д., с сохранением форматирования и стиля.

Распространённые и популярные генераторы:

- https://gohugo.io/
- https://nextjs.org/
- https://jekyllrb.com/
- http://octopress.org/
- https://hexo.io/
- https://www.gatsbyjs.org/
- https://www.mkdocs.org
- https://peachdocs.org
- https://docusaurus.io/
- https://brunch.io/
- http://presidium.spandigital.net/
- https://antora.org/

> Presidium, интересен интеграцией и импортом из Swagger, Javadoc, JSDoc (ниже они описаны).

> Ещё один продукт, на который стоит обратить внимание – Antora. Позволяет генерировать статические сайты с документацией сразу из нескольких репозиториев. Правда работает с форматом AsciiDoc.

Confluence

Сайт, документация, плагины.

Прекрасный продукт компании Atlassian, который представляет собой систему управления знаниями и документами.

Идеальный вариант для ручного ведения документации. Огромное количество плагинов, образцов контента, с которым можно работать. Полная интеграция с Jira. Генерация отчётов из Jira.

Все прекрасно, кроме возможности автоматизировать и прожорливости системы.

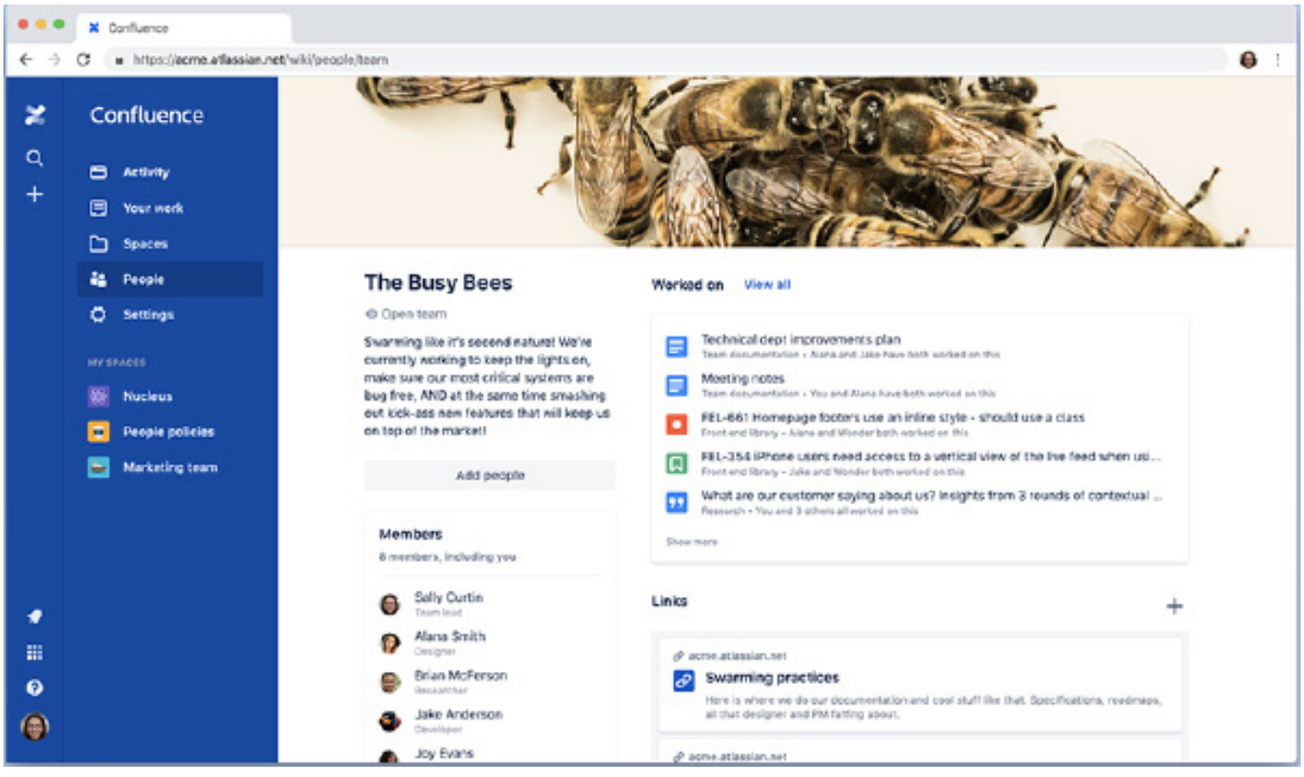
Приходится расширять плагинами.

Идеально для DocOps. git push *.md в репозиторий + cli interface Confluence (или

через Confluence API) + Документ в Confluence

Изначального такого функционала нет, но есть плагин, который позволяет делать огромное количество операций из командной строки – Confluence Command Line Interface (CLI).

Также есть ещё один полезный плагин, который позволяет импортировать markdown прямо из git – Markdown Extensions for Confluence.



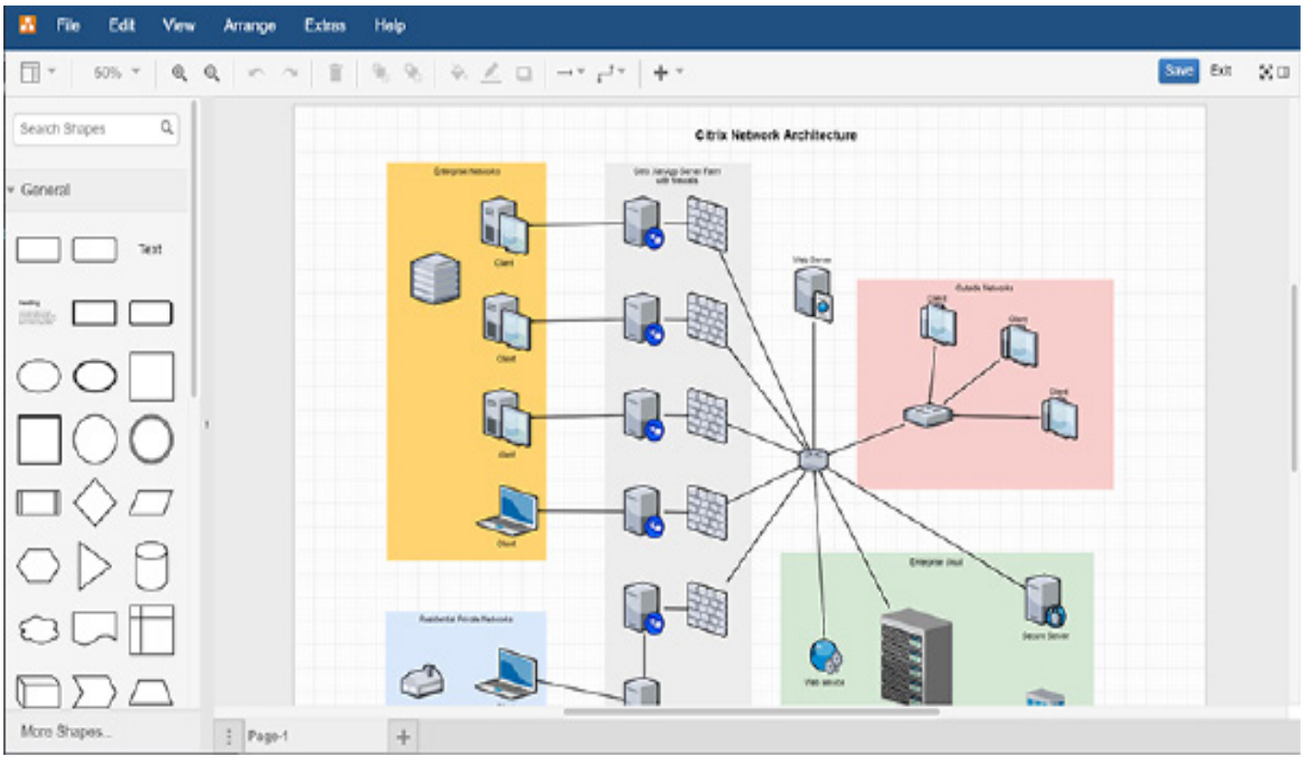
Диаграммы

Рисунки и схемы являются неотъемлемой частью любого документирования и описания ИТ-систем. Есть возможность редактировать простые диаграммы с помощью обычного текстового редактора с помощью Mermaid.

Позволяет рисовать организационные диаграммы, схемы алгоритмов и диаграммы Ганта.

Работать с ним можно такой-же схеме DocOps, которая была описана выше.

Для создания сложных схем кроме Draw.io или Visio пока ничего не придумали.



Документирование кода

Общий принцип заключается в том, что разработчик пишет комментарии в коде, используя определённые теги в зависимости от языка программирования.

После этого парсер кода генерирует документацию на основании комментариев и экспортирует её в статический сайт html.

Doxygen – самый мощный инструмент. Поддерживает парсинг C++, Си, Objective-C, Python, Java, PHP, C#. Может автоматически генерировать ссылки на официальную документацию.

Swagger – инструмент для работы с HTTP API. Создание интерактивной документации и тестирование API.

Javadoc – генератор документации в HTML-формате из комментариев исходного кода на Java от Sun Microsystems.

Jsdoc – генератор документации в HTML-формате из комментариев исходного кода на JavaScript.

Выше мы упоминали **Presidium**, который может импортировать и встраивать данные из Swagger, Jsdoc, Javadoc.

Таким образом, если команда будет придерживаться правил оформления и комментирования кода, DocOps может предложить простой, но мощный способ автоматической генерации документации.

CMDB

Что это и зачем надо?

CMDB это единый каталог, содержащий информацию обо всех ИТ-объектах сервиса и продукта и связях между ними, пришёл (пока ещё массово не пришёл, но обязательно должен) в DevOps со стороны системных администраторов (вернее ITIL).

Любой процесс техподдержки строится на грамотной CMDB и на культуре её обновления. Без CMDB о нормальной и профессиональной техподдержке сервиса можно забыть.

Какие объекты надо вносить в CMDB?

1. Все, что можно пощупать руками: сервера, роутеры, жёсткие диски, RAID-контроллеры.
2. Все, на чём работает код: VDS, instances облачных провайдеров, сервисы облачных провайдеров, ip-сети, RAID-массивы
3. Все, с помощью чего работает код: операционные системы, ssl-ключи, серверное ПО, версии ПО
4. Все, чем можно управлять инфраструктурой: логины, пароли, ключи SSH
5. Связи и зависимости между вышеописанными объектами

Естественно, это все в общих чертах и поверхностно, у каждого продукта или компании свои критерии важности того или иного объекта.

Хорошая CMDB позволяет импортировать данные или собирать информацию с агентов. Современные системы управления конфигурацией типа Ansible или Salt позволяют без особых затрат инвентаризировать хосты, на который есть доступ потом можно передавать эту информацию в базу CMDB.

Инструменты CMDB.

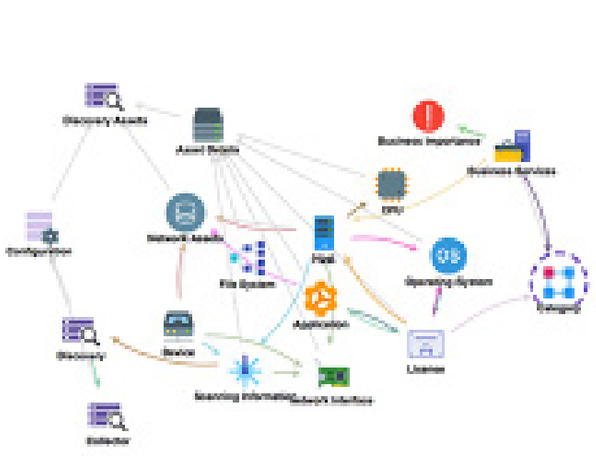
В контексте ПО, которое описано в этой книге, для CMDB очень хорошо подходит связка Jira + plugin Insight - Asset Management. Он позволяет импортировать данные из разных форматов (csv, json), подключаться к разным источникам данных (SQL). Огромным плюсом есть возможность строить разнообразные связи между объектами CMDB.

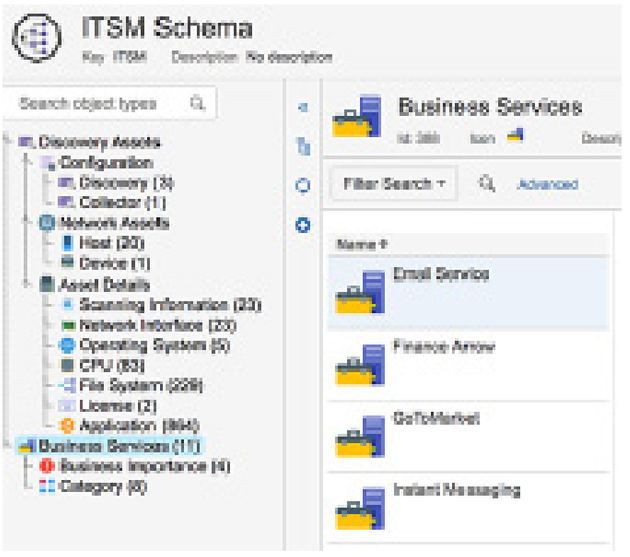
Полностью интегрируется с Jira и Confluence. Можно прикреплять объекты из Insight к задачам в Jira, вставлять в Confluence отчёты по определённым фильтрам в CMDB.

Insight Discovery – модуль для сканирования инфраструктуры и добавления в CDMB.

Также есть прекрасная возможность интегрировать Insight с AWS с помощью плагина Insight AWS Integration.

Таким образом получается очень мощное решение по работе с инфраструктурой и

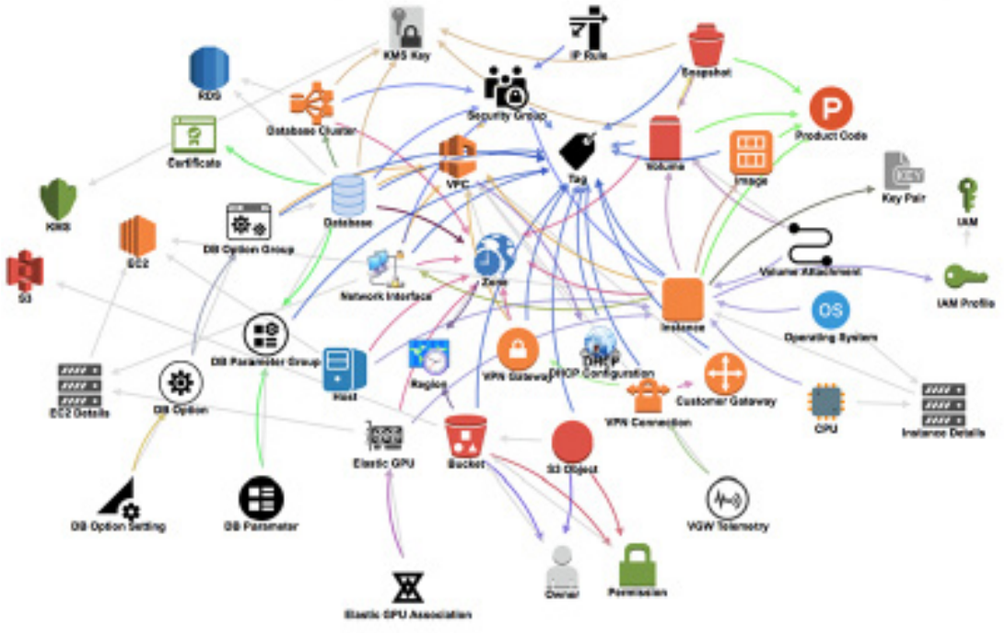




интеграцией в бизнес процессы компании.

Резюме

Без системного подхода, который максимально продуман и автоматизирован доку-



ментацию создавать не имеет смысла:

- если она устарела, то принесёт больше вреда чем пользы
- если её сложно писать и поддерживать, то затраты будут намного превышать пользу от неё
- если она не проверяется, и не исправляются ошибки, а они будут всегда, то использовать её нельзя и время потрачено впустую

Со временем команды понимают, что надо вводить в процесс технического писателя. И наличие концепции и механизмов DocOps сделают этот процесс намного эффективней.

DevSecOps

Одно из «нововведений», которое давно уже «просилось».

В некоторых процессах разработки by design предусмотрено тесное взаимодействие со специалистами по безопасности.

Нет, это не «агенты смиты» в костюмах, очах и с наушником ухе. Это мощные технари (но это не точно), которые в курсе всех открытых уязвимостей во всём софте в мире (абсолютно не точно). Их задача, кроме влияния на дизайн приложения и построения политики безопасности – следить за уязвимостями и рисками.

В связи с этим, на сцену вы ходит концепция **Application Security**:

- Проверять сканерами уязвимостей приложение перед доставкой его в production
- Держать руку на пульсе уязвимостей как на уровне технологии, так и на уровне

компонентов и сторонних библиотек, которые используются при разработке
- Периодически проводить тесты на проникновение (если разрабатывается сервис)

Главная задача не обнаружить уязвимость, а предотвратить её появление. Поэтому нашего условного «безопасника» надо подключать почти на всех стадиях разработки.

Вплоть до лекций перед разработчиками о том, как написать безопасный код.

Но тут крылся один изъян – тестирование безопасности не успевает за скоростью разработки. Мы же помним, что DevOps заточен именно на скорость разработки, вся автоматизация направлена именно на ускорение процессов тестирования, доставки приложений.

Ручная проверка на стандарты безопасности очень долгий и дорогой процесс, который использует дорогие инструменты. Раз в месяц «безопасник» сканирует код на предмет уязвимостей, делает разные pen-тесты, составляет отчёт и презентует его разработчикам. Мы же знаем, что чем больше страниц в отчёте, тем весомей выводы.

> Начинается следующий диалог:
> -- Вы кто? (Если человек появляется раз в месяц и от него не зависит зарплата, то его, как правило, углублённые в код разработчики не запоминают)
> -- Специалист по информационной безопасности проекта
> -- Что вы даёте? Что это за кипа бумаг (что это за PDF на 150 страниц вы мне прислали?)
> -- Это подробный отчёт по уязвимостям, которые показало тестирование
> -- Тестирование чего?
> -- Проекта
> -- Почему вы даёте отчёт именно мне? Я занимаюсь фронтэндом. Это все ошибки в моем модуле?
> -- Нет. Это полный отчёт по всем модулям.
> -- Ну хорошо. Мы посмотрим. Спасибо.
> -- Распишитесь в получении отчёта.

Понятно, что диалог придуман, но при классической модели, как правило, так и происходит.

Поэтому возникла необходимость как-то ускорить «безопасников» и интегрировать их в общий рабочий процесс. Так появился **DevSecOps (Develop + Security + Operations)**.

> Как уже понятно, DevSecOps это тоже процесс. И как любой процесс он должен быть согласован, понят и принят всеми. Надо сказать, что именно DevSecOps сейчас фаворит среди всего «девоповского» движения.

Просто достаточно много компаний заявляют об утечках данных пользователей (а ещё больше не заявляют, когда они происходят) и бизнесу начинает немного надоедать терять деньги.

Поэтому появилось такое направление как **Security Development Lifecycle**, основоположником которого стала компания Microsoft. В дальнейшем эта модель была конкретизирована и разбита на разные методологии – **OpenSAMM, BSIMM, OWASP**.

Также, при переходе безопасности на рельсы DevOps начали меняться и рабочие процессы. Стала появляться такая себе «волонтёрская» роль **Security Champions** – это человек или группа людей внутри команды разработки, которые заинтересованы в безопасности продукта.

Какие у них роли и функции:

- Они работают обычными разработчиками и могут быть посредниками между классическими «безопасниками» и командой. Своего рода евангелисты безопасности. Им доверяют, они свои для команды, их доводы по включению элементов тестирования безопасности кажутся команде разумными. Отчёты на 150 страниц никто не читает.
- Они разбираются или хотят разобраться со стеком инструментов по тестированию безопасности в разрабатываемом продукте. Со временем они учатся его профессионально и эффективно использовать.

- Они знают код, проблемы, планы именно своего проекта. «Безопасник» может быть один на всю компанию с дорогущим сканером в руках, и он естественно особо не вникает в тонкости. «Вот отчёт, в нем список возможных уязвимостей, обратите внимание на этот модуль, я не дам добро на развёртывание в production, у вас потенциальная дыра». Хотя этот модуль вполне может быть в планах на удаление в самом проекте, но совсем по другим причинам и в production его не планировали.
- Со временем они начинают консультировать коллег по вопросам безопасности. Естественно это не посылание в google и не цитирование страниц отчёта «агента смита». Они знают наиболее «дырявые» библиотеки в используемых языках программирования, они уже в состоянии подобрать необходимые паттерны программирования и проектирования. Могут делать ревью кода для коллег.
- Через какое-то время они могут проводить небольшие тренинги.
- Самое главное – они постепенно участвуют в трансформации и переводке разработки на рельсы DevSecOps

Зачем им это надо? Они растут профессионально, при таком подходе они учатся убеждать коллег и постепенно развивают лидерские навыки. В конце концов они начинают больше стоить на рынке.

Диапазон применения DevSecOps

Рассмотрим сферы, методы и инструменты для интеграции процессов безопасности продукта в разработке:

1. Проектирование. Какая основная «боль» у «безопасников» и соответственно, какие могут возникать проблемы уже в эксплуатации. Как их избежать на этом этапе.
2. Зависимости. Любой проект – это тонны зависимостей, которые используются при сборке и вводе в эксплуатацию. Основная беда и источник дыр в безопасности. Свой репозиторий с необходимым ПО и библиотеками.
3. Анализ кода. Насколько безопасны паттерны написания кода, сколько отклонений от Security Development Lifecycle
4. Тестирование продукта или сервиса. Тестирование на проникновение, паттерны «плохого» пользователя.

DevSecOps при проектировании

> Основной вопрос и проблема при любой разработке – управление credentials. «А какой пароль на монгу на стейдже, вы что-то там меняли?», «Я не могу попасть на бэкапный сервер», и самое страшное – «Добрый день, это маркетинг беспокоит, вышлите нам пароль на сайт, нам надо лэндинг поменять».

Утрировано, но правда. Пароли в env-файлах, API-ключи на GitHub (уже есть куча краулеров, которые сканирую репозитории и выдёргивает из plain text все самое «вкусное»). Жизнь ничему не учит.

Идём дальше.

> «У нас вся инфраструктура за firewall, во внутренней сети, зачем нам ssl на API? Обращения идут по внутренним «айпишникам». Если взломают, то уже какая разница?».
> «Надо сделать пользователя для сервиса, можно без пароля» и т.д.

Это только два момента из множества подобных ситуаций.

Что начинает делать DevSecOps в таких ситуациях как эти:

1. Описание на уровне официальных документов и политик периметров безопасности. VPN, шифрование разделов, использование SSL/TLS в эндпоинтах для

подключений. Соответственно если речь идёт о внутренней сети – использование самоподписных сертификатов и отключение проверки на валидность в разрабатываемых модулях (в некоторых сервисах надо это явно указывать)

2. Использование системы централизованного управления credentials. Никаких паролей в открытом виде в файлах, сценариях сборки и развёртывания

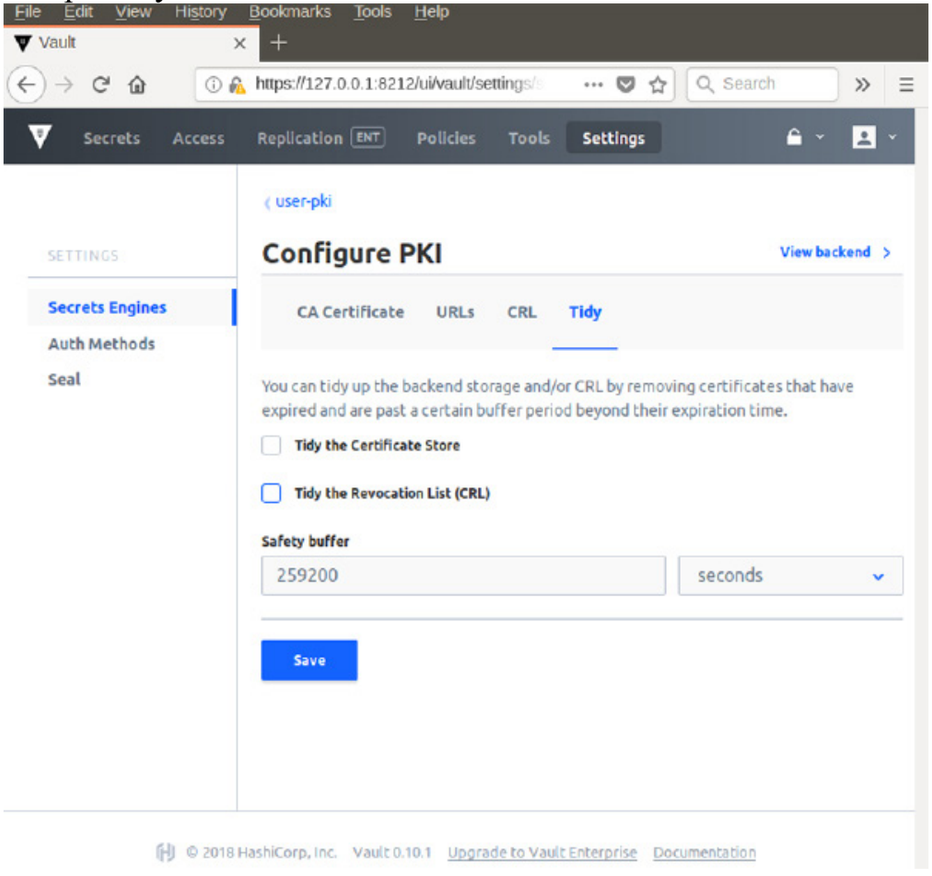
Остановимся на инструментах централизованного управления credentials. Тема актуальная и болезненная для всех.

Инструменты централизованного управления credentials.

Hashicorp Vault

Сайт, документация, варианты использования.

Самый распространённый инструмент хранилища конфиденциальных данных. Хранилище создаётся и шифруется с помощью 5-ти ~~колец всевластия~~ ключей. Для расшифровки надо как минимум три ключа. Credentials создаются в произвольном виде ключ-значение. Из credentials делается токен, который доступен по API. Для использования API и ключей надо использовать в коде и сервисах используется vault-agent. Токены периодически перевыпускаются.



Инструмент достаточно мощный и может использоваться в разных вариантах вплоть до шифрования разделов.

Такой принцип авторизации и аутентификации надо воплощать на уровне кода. И это необходимо предусматривать ещё на этапе проектирования.

Keywhiz

Сайт.

Ещё одна система хранилища конфиденциальных данных. Состоит из следующих компонентов:

- Keywhiz server – хранит базу ключей
- KeywhizFS – виртуальная файловая система, которая создаётся при подключению

к серверу и туда загружаются необходимые ключи
- Keywhiz-cli – клиент для подключения к серверу

Документация очень скудная, но система заслуживает изучения

Teleport

Сайт, документация.

Немного выпадает из общего ряда рассматриваемых систем, поскольку является SSH-бастионом. Но если учитывать, что огромное количество сервисов хостится на Linux/BSD системах, то заслуживает изучения и использования.

Возможности:

- Security gateway для SSH или SSH-over-HTTPS через браузер (прощай Putty)
- Security gateway для кластеров Kubernetes
- Позволяет записывать SSH-сессии
- Kubectl exec recording/replay
- Web-интерфейс
- Role-based access control (RBAC) для SSH соединений
- Поддержка 2FA для SSH и Kubernetes
- Использует основанный на сертификате доступ с автоматическим истечением срока действия сертификата

Secrets

Сайт.

> Как раз для варианта «Добрый день, это маркетинг беспокоит, вышлите нам пароль на сайт, нам надо лэндинг поменять».

Можно (и нужно) развернуть на своём сервере. Очень просто принцип действия:

1. Создаётся credentials
2. Указывается сколько минут будет жить ссылка, которая сгенерируется
3. Указывается пин-код для открытия ссылки
4. Генерируется уникальная ссылка. Для получения credentials надо ввести пин-код.

Дёшево и сердито.

DevSecOps и зависимости

По большом счету вся безопасность в ИТ очень призрачная.

По причине того, что системы настолько усложнены и разрабатываются огромным количеством людей, вероятность уязвимости просто огромная. Любая система, начиная от Linux и заканчивая Node.js, использует пакеты и модули, которые написаны разными людьми.

Если описывать open source системы, то есть ещё огромный риск того, что модуль или пакет уже не поддерживается разработчиком. Ну бывает – не интересно и человек решил нужную ему задачу, не хватает времени, не согласен с сообществом, хочет денег или пожертвований и свернул разработку.

Причин море. Но пакет или модуль уже есть в зависимостях. И таких зависимостей могут быть сотни на одном проекте.

Развиваем тему дальше. Огромная часть зависимостей это open source, у которого есть огромное количество лицензий. И некоторые из них почти прямым текстом говорят, что код, который создан с использованием этих зависимостей тоже должен быть впоследствии открыт.

Другими словами, если вы используете мою библиотеку и на её основе что-то напи-

сали, то давайте делитесь.

> Наверное, некоторые заказчики будут несколько удивлены тем, что код, за который они заплатили надо предоставить в открытый доступ.

И так, есть две проблемы, которые необходимо решить и ещё один пункт, который надо предусмотреть в разработке:

1. Поиск уязвимостей, которые используются при написании кода и сборке проекта. Неподдерживаемые пакеты. Пакеты, в которых есть незакрытые уязвимости.
2. Лицензионная чистота зависимостей.
3. Отдельный репозиторий с пересобранными пакетами. Часто «безопасники» требуют пересборку пакетов с нужными параметрами или пересборку пакетов из исходных кодов, поскольку сопровождающие дистрибутива не выпустили обновления, которые закрывают уязвимость. Так называемые 0-day уязвимости (уязвимость «нулевого дня», когда о ней объявлено, но официальных «заплаток» ещё нет)

Понятно, что вручную всё это отслеживать нереально. Тем более нет смысла делать это раз в месяц (как было описано вначале главы) – это породит регрессии в коде проекта и придётся много переписывать. Поэтому необходимо автоматизировать процесс, интегрируя средства автоматической проверки в CI/CD.

Из открытых продуктов очень мало инструментов, которые можно использовать для этих целей, но что-то найти можно. К сожалению, мы не сможем подробно описать функционал и возможности – тема очень сложная и объёмная.

OWASP Dependency-Check

Сайт, документация.

Поддерживаются следующие языки программирования: Java и .NET. В экспериментальной стадии поддержки Ruby, Node.js, Python.

Есть плагины для подключения к Jenkins.

DevSecOps и анализ кода

Следующий вопрос – как помочь разработчику писать код, но при этом анализировать его на ошибки. Разливают два подхода.

Статический анализ – тестирование не в среде исполнения, проверка кода программы на все возможные архитектурные недочёты.

Динамический анализ – тесты проводятся во время работы программы, после сборки и deploy

Инструменты статического анализа

SonarQube

Сайт, документация.

Это мастодонт и стандарт де-факто для автоматического тестирования.

Поддерживает более 25 языков программирования. Интегрируется с системами CI/CD.

Есть возможность проверять правильность написания кода и уязвимости в зависимостях сборки. Поддерживает мульти язычные проекты (когда проект написан на разных языках программирования).

Crucible

Сайт, документация.

Продукт Atlassian, который естественно интегрируется со всей линейкой продуктов.

Позволяет анализировать код, обсуждать его в виде чатов.

> Список дорогих коммерческих решений, специально не рассматриваем, поскольку такие продукты требуют обработки огромного количества материалов и тестирований. Просто вы должны знать, что такие продукты есть и их можно купить при необходимости.

- PVS Studio
- JFrog Xray
- Coveryity
- Fortify
- Checkmarkx
- Veracode

Open source проекты для статического анализа кода

- C/C++ – flawfinder
- Python – bandit
- Ruby on Rails – brakeman
- Java – find-sec-bugs
- Go – Gosec
- PHP – phpcs-cecurity-audit
- .NET – Security Code Scan
- Node.js – NodeJsScan

Open source проекты для динамического анализа кода

OWASP ZAP

Сайт, документация.

OWASP Zed Attack Proxy (ZAP) является одним из наиболее популярных бесплатных инструментов для сканирования безопасности и активно поддерживается сотнями международных добровольцев.

Помогает автоматически найти уязвимости в системе безопасности веб-приложений во время разработки и тестирования. Также является отличным инструментом для ручного тестирования безопасности.

Arachni

Сайт, документация.

Arachni это бесплатный сканер безопасности с открытым исходным кодом, который может быть использован для создания автоматизированных отчётов о безопасности для веб-сайта.

Существует широкий спектр проверок безопасности, которые могут быть индивидуально выбраны для включения в сканирование, включая XSS (с DOM варианты), инъ-

екции SQL, NoSQL инъекции, инъекции кода, включение файлов и т.д. на всех популярных платформах.

Arachni включает в себя интегрированную среду браузера для того, чтобы обеспечить достаточный охват для современных веб-приложений, которые используют технологии, такие как HTML5, JavaScript, манипуляция DOM, AJAX и т.д.

DevSecOps для тестирования продукта или сервиса

Это последний этап проверки, который уже не включает автоматизацию. Тут как раз приходит время «агентов смитов». Которые берут в руки профессиональные сканеры и вообще начинают себя очень плохо вести по отношению к детищу разработчиков.

В корне меняется подход к тестированию.

> В основном, разработчики тестируют продукт в ключе «положительной логики» (happy path). Подразумевается, что пользователь ведёт себя хорошо и правильно. «Безопасник» должно тестировать в ключе «грустного пути» (sad path) – подразумевается, что пользователь ведёт себя как мошенник и взломщик. Иногда это может привести к совершенно неожиданным результатам.

Он может заполнять поля неправильными данными, и окажется, что форма регистрации не имеет проверок на валидность данных. Можно атаковать API валом запросов и положить какой-то сервис.

Можно вычислить IP площадки и начать сканировать порты или устроить небольшую DDoS атаку (это мы сейчас намекаем, что всё должно быть спрятано за DNS того же Cloud Flare). Много чего можно делать плохого, о чём не подозревают разработчики.

Резюме

Как видите можно очень красиво включить проверку на безопасность в процесс разработки.

Это огромная тема для изучения, тестов и самообразования. Реально делать безопасный продукт с «минимумом» усилий. Да, придётся несколько изменить подходы и не отмахиваться от этих непонятных людей с их отчётами. Зато потом не придётся краснеть перед инвесторами или лихорадочно придумывать причину взлома или деградации сервиса.

Как вы понимаете, это никого особо волновать не будет. Ну и естественно можно продать свой продукт намного дороже.

> P. S. Очень рекомендуется досконально изучить все рекомендации и продукты OWASP™ Foundation, особенно раздел все продуктов: **«Flagship Projects»,** там очень много интересного.

Процессы в DevOps

Начнём с самого интересного и одновременно спорного.

Тезисы:

1. У каждой из сторон процесса своё мнение что такое DevOps. И каждая из сторон по-своему права. Классический вариант притчи о трёх слепых и слоне.
   «Девы» считают, что это «автоматизаторы», они бы и сами все это сделали (а в некоторых местах так и происходит), но они «оверскилл» для этого баловства, плюс это не интересно да и времени нет.
   Очень показательный и полезный пример на эту тему – вебинар «Как деплоить в прод по многу раз в день и [почти] ничего не ломать».

   Докладчик акцентирует внимание на том, что у них «правильный девопс» – раз-

работчики сами управляются с инфраструктурой. Справедливая точка зрения, но стоит учесть, что компания не очень большая, разработчики уровня middle и senior. Интересно как они будут масштабироваться за счет juniors, или отдельной службой поддержки первой линии.

«Опсы» справедливо считают, что «девопс» это тот чувак, который занимается этими всеми «деплоями», «А/Б-тестами» («деплой» это такая штука после которой вечно начинается какой-то «геморрой» на «продакшене»). Да и вообще, он больше знает про архитектуру сервиса. Если что – у него надо спрашивать.

2. Agile, на основе которого возник Scrum и тот же DevOps; и не связанные с ним совершенно самостоятельные вещи как ITIL, CRM – это методологии, другими словами набор правил, которые надо применять в определённых ситуациях.
   Это не софт, хотя термин «CRM-системы» прочно закрепил это заблуждение.
   Соответственно появилось ещё одно заблуждение под названием «DevOps-инженер».
   На уровне правильных определений – DevOps-инженеров нет, а на практике только они и есть. Кто-нибудь встречал Agile-инженера? Есть человек, который владеет методологией DevOps и соответствующими инструментами.
   Хотя… пусть будет DevOps-инженер, термин уже раскрученный. Как говориться – «все пиар, кроме некролога».

3. DevOps – тоже относится к бизнесу. **“Time to market”**. Это означает, что DevOps также должен участвовать в трансформации бизнеса, а не только разработки и операционной поддержки.
   Главная цель, которую все ждут от «DevOps-инженера» это автоматизация. Автоматизировать хаос невозможно, значит надо хаос победить, или минимизировать, или отодвинуть от себя на максимальное расстояние.

> Поэтому мы будем описывать процессы взаимодействия с разработчиками, с поддержкой и с бизнесом. Если где-то в этих направлениях будет дисбаланс или непонимание, то все методологии и «лучшие практики» можно спокойно положить в тумбочку и настраивать CI/CD и Kubernetes за все деньги мира.
> Но не долго. Поскольку через некоторое время у команды будет работающий CI/CD, Kubernetes и ELK, только не будет больше потребности в дорогом «девопсе», всё ж работает. Если вся эта конструкция начнёт заваливаться, то можно опять «нанять девопса».

Общее для всех примечание – DevOps надо «продавать» и внедрять. Каждому клиенту (разработке, эксплуатации, бизнесу) по-разному.

Мы думаем, что скоро модель DevOps вообще разовьётся в отдельный сервисный процесс в компаниях (и мы будем очень этому рады), со своими метриками, регламентами и т.д. Это хорошо видно на примере ITIL.

DevOps и разработка

Взаимодействие с командой в рамках DevOps можно разделить на несколько видов:

1. Стандартный рабочий процесс – заявки от разработчиков, регламенты по deploy и т.д. Сюда же включаем стендапы, спринты, ретроспективы.
2. Обучение – выяснять на уровне всей команды пробелы по технологиям и инструментам проекта, у всех должно быть одинаковое представление о схеме и принципах работы приложения или сервиса.
3. Внештатные ситуации – инциденты: в production и в рабочей инфраструктуре, DDoS-атаки, утеря данных, саботаж.

У каждого вида свой подход, процессы и инструменты.

Стандартный рабочий процесс

Останавливаться сильно подробно нет смысла. Использовать Scrum или другое ответвление Agile-методологии.

> Ещё раз напомним – DevOps в команде стоит рассматривать как сервис со всеми вытекающими из этого процессами: **SLA (Service Layer Agreement)**, отдельные проекты, тайминги для типовых действий, приоритеты заявок и т.д.

Надо понимать, что стандартный рабочий процесс DevOps связан с деньгами, особенно когда используются облачные провайдеры – поэтому некоторые заявки могут требовать от разработчика обоснований и утверждений у финансового руководства.

Что от нас будут хотеть разработчики?

- Окружения – как можно быстрее, по запросу
- Помогать со сборкой при CI/CD если что-то пойдёт не так
- Хороший мониторинг, который будет автоматом поднимать упавшие сервисы или приложения
- Автоматизировать развёртывание до блеска на все среды в том числе и на production
- Сделать так, чтобы они могли посмотреть log-и в любой момент времени на любом окружении
- Перекинуть на nginx или haproxy в нашем лице решение какой-нибудь проблемы, которую долго решать кодингом
- Рассказать им что и как работает на production или вообще, хватит ли мощности
- Оставить на них только ответственность за код и рефакторинг, остальное не должно их волновать
- Не мешать и не мельтешить перед глазами

Основные процессы, которые можно выделить и грамотно построить для более эффективной работы.

Доставка кода (deploy)

В основном будет идти речь про web-сервисы, но каким-то образом к простым приложениям его тоже можно применять.

Что надо понять в разрезе deploy:

- Поделить их на типы со своим уровнем риска. Например, «косметические изменения», «откатить невозможно», «релиз», «архитектурные изменения». Соответственно под каждый тип надо продумать уровень автоматизации с помощью веток, тегов в git. «Косметические изменения» могут отправляться в production окружение автоматически, «архитектурные изменения» – только вручную. И т.д. Каждый член команды должен чётко понимать разницу в этой терминологии.
- В зависимости от уровня риска, продумывать, когда «деплоить» и какие сотрудники должны присутствовать. «Задеплоить» «архитектурные изменения» в 20:00, а потом в 04:15 выяснить от техподдержки, что под нагрузкой все очень плохо – распространённый сценарий «выстрела себе в ногу». В таком режиме через какое-то время начинается «текучка», нервотрёпка и депрессия у сотрудников.

С одной стороны, это очень просто написать, как нам в книге. С другой, очень сложно объяснить «бизнесу», что «фишку», которую маркетинг заказал давно, которую мучали в разном виде на протяжении 8 спринтов, которая «ну вот же блин работает!!!», выкладывать в production будут в 11 утра следующего дня. А сейчас все идут отдыхать, завтра может быть тяжёлый день.

Наверняка у каждой команды есть своя «боль» на эту тему.

Технический регламент

Это общее название для совокупности таких процессов как:

- Onboarding нового разработчика
- Техподдержка разработчиков
- Системы ветвления Git
- Поведение при «поломках» сборки в CI
- Работа с техническим долгом
- Технологический стек разработки

Рассмотрим более подробно.

Onboarding нового разработчика

Как правило, этим занимается HR, но есть ряд вопросов, на которые могут ответить только коллеги.

Какая платформа, где «хостимся», какие версии, как проходят автотесты, как устроен deploy, как часто должны выходить релизы, какая система ветвления используется (почему нельзя использовать ту, к которой привык), какие сейчас есть проблемы (технический долг) и т.д.

В идеале, новичок получает рабочую машину, credentials, скрипт, который развернёт с помощью Ansible все рабочее окружение, включая IDE с плагинами и настройками (а их должно быть достаточно, если вы прочли предыдущие разделы).

Короче, при onboarding новый сотрудник должен получить все ответы на вопросы из списка выше, изучить их и подтвердить своё понимание. Растягивать все это на месяц, задавая вопросы по мере поступления точно не стоит. Пусть лучше почитает неделю и задаст все вопросы сразу. И заодно пройдёт мини-аттестацию на эту тему. Заодно можно увидеть в какой области он «плавает». И самое главное – «бизнес» должен ему объяснить несколько важных вещей:

- Почему клиенты покупают продукт, к созданию которого он будет причастен («по- гуглите» концепцию “jobs to be done”)
- За что клиенты платят деньги
- Почему клиенты выбирают именно этот продукт
- Какие есть конкуренты у продукта

> Не обязательно «зажечь» новенького «миссиями компаний и взрывом рынков» – не все болеют за общее дело, но делать вид, что он просто «ждёт таску в джире» за пару штук долларов в месяц ему будет трудно.
> Карты перед ним раскрыли, общую цель наметили и ругать «тупой маркетинг» уже будет труднее. Тут ещё главное, чтобы маркетинг получил такую же информацию.

Техподдержка разработчиков

Как мы уже говорили выше – DevOps лучше применять как отдельный сервис. Это может звучать излишне бюрократично, но именно такой процесс понятен обеим сторонам.

Почти все когда-то были (в начале карьеры) связаны с ИТ-услугами как со стороны клиента, так и со стороны поставщика услуг. Поэтому все должны примерно понимать, как будет обрабатываться их заявки в рабочих чатах, почему ведётся уточняющая переписка и т.д. Почему сразу получить желаемое не получится.

Плюс у самих разработчиков могут быть проблемы на рабочих машинах – не работает VPN, MacOS (Windows) после обновления шалит, какие-то проблемы с IDE.

Идеальный вариант для таких случаев – пересаживать на подменную машину и накатывать Ansible или аналогами все необходимое. Плюс, зная тайминги (или SLA) по заявкам разработчики могут лучше планировать свою работу.

Системы ветвления Git

В очередной раз поднимаем эту тему. С одной стороны, она достаточно простая, с другой – архиважная. Особенно если CI/CD создаётся для нескольких команд. Flow должен быть одинаковый для всех.

Сейчас есть огромное количество методик разработки, некоторые описаны в этой книге, например, Feature Branching. Выбор методики очень дискуссионный вопрос, мы не можем что-то советовать **(да блин, можем – давайте все использовать GitLabFlow!!!)**, каждая команда решает для себя, что ей больше подходит.

И уже на основе этого надо подбирать инструменты для CI/CD. Лучше обо всех этих методиках договориться ещё «на берегу», до старта внедрения.

Поведение при «поломках» сборки в CI

Раз уж мы «девопсы» и нас, нанимают для «CI/CD, Kubernetes и «канарейка», то надо чётко обозначить команде что с этим CI делать и с как его правильно «готовить».

Есть задача –тестировать и собирать код как можно чаще. Хорошо, когда он собирается, тестируется без проблем. CI работает, тестировщики (или Selenium) «прокликивают», маркетинг или бизнес смотрит и формирует свои замечания.

Что делать, когда не собирается? Вроде диверсий никто не устраивал, сознательно не «говнокодил». Особенно, когда на машине разработчика все работает, а на test нет.

Возникает несколько вариантов:

- Разные версии софта и библиотек у разработчика и у тестового окружения. Дело в том, что operations, да и DevOps это они с виду ничего не делают, а на самом деле очень даже работают по своим регламентам – обновления, патчи и т.д. Иногда это даже автоматизировано (и вот это очень плохо – парадокс). В таком случае надо или общий для всех образ Docker (что уже скоро будет в 99% случаев), или список версий, и чёткий регламент на обновления, в том числе и разработчику.
- Все одинаковое, но локально работает, а на test проваливаются тесты. Значит на http://localhost все хорошо, а на https://test.company.com все плохо. Оказывается, https://test.company.com это Cloud Flare proxy + SSL + haproxy + nginx бэкенд, и на каждом из этапов используются WAF и разрешённые HTTP-методы. Ну тут стандартная «вилка» из «а чего ты не спросил, а чего ты не сказал». Читаем выше про onboarding новых сотрудников.
- Все везде одинаковое, но «нит». Смотрим логи, разбираемся, помощь зала.

Как видите, первые два пункта, при правильном DevOps подходе вообще не должны возникать. Третий пункт и действия по его нейтрализации решаются каждой командой по-своему.

Понятно, что есть приоритеты для «починки конвейера», не закрытая скобка на «фронте» и ошибка в бэкенде это разные вещи, как правило это отдельные git-репозитории и для них можно устанавливать разные приоритеты по «починке».

Второй этап – полная унификация инфраструктуры через IaC. Даже крамольную вещь вбросим – но лучше вообще каждому разработчику дать по своему окружению на test, а не отлаживать «сырой» код локально на машинах. В любом случае, про «поломанную» сборку узнает DevOps и пытается чинить тоже он.

Работа с техническим долгом

Технический долг будет всегда. Никто не совершенен, плюс не забываем, что работаем по Agile и требования «бизнеса» постоянно меняются. Поэтому технический долг будет.

Другой вопрос, что при большой текучке разработчиков, при плохой работе с Jira и документированием процесса технический долг может плавно перекочевать в разряд местных «городских легенд» отдельно взятого сервиса.

> «Всегда перезагружай Redis перед Nginx, смотри не перепутай», «мы никогда не обновляем java… нигде», «эта виртуалка должна быть всегда включена».
> Все эти «мифы» постепенно наслаиваются один на другой и через какое-то время надо уже нанимать экзорциста, а не разработчика.
> Самое интересное, что подобные мифы рождаются именно в отделе поддержки (или operations), разработчики как мудрые жрецы посмеиваются над необразованными крестьянами… но у них табу на обновления npm, а вот это серьёзно и грозит анафемой за «поломанный прод».

Что с этим делать?

Фиксировать и признавать, что это долг и он должен быть «погашен». О нем должны знать все, в том числе и «бизнес», если технический долг влияет на продукт.

Все участники со стороны разработки и эксплуатации должны всеми возможными легальными способами требовать время на погашение всех долгов (что-бы потом успешно создать новые). Он должен быть чётко задокументирован.

На практике, уже хорошо, когда про технический долг знают хотя бы все «технари». А вот дальше все очень грустно и все ждут, когда «все перепишем на Go».

Технологический стек разработки

Выше мы немного упоминали влияние версий и набора софта на разработку. В принципе, было бы прекрасно все это оформить в отдельный документ, можно назвать его «технологическая карта».

В этом документе можно (нужно!!!) очень чётко и подробно описать все, что относится к сервису или приложению:

Разработка:

- IDE
- Плагины
- Образы Docker (доступ к приватному registry при необходимости)
- Credentials
- Git репозиторий с provisioning для разработчиков

Окружения:

- Общая схема dev, test, stage, production
- Базы данных – версии, система репликации и шардинга (при необходимости)
- ОС – версии ядра, настройки sysctl
- Список софта, который используется в построении сервиса (Nginx, Haproxy, Kubernetes etc) с версиями
- Эндпоинты для бэкендов
- Особенности сборки Docker образов для всех окружений (если используются свои версии образов в private registry)

Процесс:

- Схема доставки кода
- Схема резервного копирования и восстановления, приблизительные тайминги восстановления на production
- Распространённые типы ошибок и соответствующая эскалация

Это только общие черты документа, естественно, каждая команда может создать свой, но идея понятна.

Обучение

Что даёт обучение в команде:

- Растёт общий уровень знаний команды. Это позволяет всем дальше развиваться немного быстрее. Скорость эскадры равняется скорости самого тихоходного корабля.
- Растёт уровень понимания архитектуры продукта, что ведёт к пониманию слабых мест в архитектуре, которые необходимо исправить.
- Начинается более активное внедрение технологических преимуществ у продукта. Все начинают понимать, как устроены «зубры» рынка, как они работают под высокими нагрузками.
- Уменьшается «фактор автобуса». Не должно быть человека, который единственный понимает, как все работает.
- Дальнейшая разработка ведётся с точки зрения намного более широкого кругозора, понимания реального уровня и стоимости ресурсов провайдеров. Иногда даже начинается проектирование с расчётом затрат/прибыли.
- Новые сотрудники намного быстрее начинают работать в полную силу.

Что изучать, в каком объёме.

- Технологии, которые используются в проекте. В проекте всегда используются DNS, SNAT/DNAT, SSL, HTTP, маршрутизация и т.д. Базис, на котором все работает.
- Изучать документацию по сервису. Как устроен production. Во время внештатных ситуаций необходима быстрая реакция от всей команды. Времени на чтение уже не будет.
- Изучать основные алгоритмы работы сервиса. Что происходит при каких событиях, куда идут запросы, куда и что записывается, где хранятся сессии. Причём это надо изучать больше службе эксплуатации.

Это минимум, который позволит выровнять экспертизу в команде и даст возможность хоть в какой-то степени подстраховывать друг друга.

DevOps и эксплуатация

Вечная проблема «мозгов» и «рук», у которых инцидент в общей зоне ответственности.

> Что-то не работает. Или работает не так как надо… и понеслась.
> -- Что вы там опять понаписывали??!?
> -- У нас все работает, смотрите у себя…

Знакомо?

Какие процессы можно выделить при соприкосновении «девопса» со своими «дальними родственниками» из эксплуатации (поддержки):

- CMDB, документация и чек-листы
- Правила эскалации заявок

CMDB, документация и чек-листы

При DevOps подходе вышеописанная ситуация «у нас, у вас» плавно исчезает. Конфигурация должна быть везде одинакова, во всех окружениях. Разная может быть только нагрузке.

При DevOps подходе должна быть актуальная документация по всем важным аспектам продукта или сервиса. «Отморозиться» эксплуатации уже не удастся, что не предупреждали, не описано в документации.

Процессы в поддержке (эксплуатации) уже давно описаны и отработаны в ITIL – Стратегия услуг (англ. Service Strategy), Проектирование услуг (англ. Service Design), Преобразование услуг (англ. Service Transition), Эксплуатация услуг (англ. Service Operation), Постоянное улучшение услуг (англ. Continual Service Improvement).

Это тоже методология, она тоже внедряется через «нам такое не надо, у нас все хо-

рошо», там тоже нужны грамотные «внедренцы» и «продавцы» инициаторы.

Из инструментов есть Jira Service Desk, который все процессы методологии более-менее поддерживает.

Выше (в разделе «Документация») говорилось, что необходимо прийти к пониманию терминов ITIL на уровне разработки и эксплуатации – инциденты, запросы на изменения и т.д. должны иметь для всех участников процесса одинаковый смысл.

Всегда в любой службе эксплуатации все упирается в два фактора, на которые она может влиять только косвенно:

- Документация
- Скорость передачи и обновления знаний и компетенций

Это если принять за аксиому, что у нас в службе эксплуатации сидят высококлассные специалисты, которые разбираются в технических нюансах продукта.

Что описывать, как описывать мы подробно изложили в разделе «Документация», там же расписано, что такое CMDB. Ещё раз повторимся, что без CMDB ни о какой нормальной эксплуатации не может быть и речи, особенно при растущем сервисе.

Поговорим о скорости передачи и обновления знаний. Это двунаправленный процесс. Предположим, что все напряглись и сделали подробную документацию по продукту, которая включает все аспекты. Может даже для этих целей внедрён DocOps.

Может даже документацию все прочитали и задали вопросы, на которые получили ответы. И даже может быть (хотя это из области фантастики) документация присутствует в процессе onboarding новых сотрудников.

Другими словами, у нас есть состояние, когда система управления знаниями устойчивая и все хорошо.

О чём часто не знает служба эксплуатации:

- **Технический долг**. Он есть постоянно, он влияет на качество продукта. Даже если он не выявлен или не признан разработчиками и таковым не называется. Возможно такая ситуация (признание) это просто вопрос времени. Соответственно (и об этом уже несколько раз упоминалось в книге), о нем должны знать все. И особенно поддержка.
  Что с этим делать. Чётко документировать и предусмотреть наличие технического долга в инструкциях и чек-листах.
- **Маркетинговая активность**. Часто про неё никто не знает. В свою очередь, она оказывает влияние на продукт в виде возросшего трафика или нагрузку на API и базы данных. Более того, маркетинг сам может не знать, что начался ажиотаж, поскольку реклама и реакция на неё могут не коррелироваться.
  Что с этим делать. Звонить в маркетинг при каких-то всплесках или требовать от них приблизительный план-календарь медиа-активности. По крайней мере у маркетологов есть прогнозы по срабатыванию рекламы и планы по «нагону» трафика.
- **Архитектурные изменения продукта**. Для разработчика и архитектора фраза «давайте использовать Apache ActiveMQ вместо RabbitMQ» говорит намного больше, чем для первой линии поддержки. Особенно, когда они не совсем понимают, что это вообще такое.
  Что с этим делать. Главное правило – эксплуатация должна быть подготовлена к изменениям. Не знать, что они произошли, а именно быть подготовлена. Новый сервис – новые чек-листы, документация. И только потом внедрять изменения.
- **Изменения в регламентах у разработки**. Стандартный спринт две недели. Поддержка об этом знает, прямо или косвенно. Потом разработчиков попросили ускориться для того чтобы успеть к срокам сдачи проекта. Начинается использование всей мощи методологий DevOps – огромное количество доставки кода на «боевые» окружения, иногда фиксы прямо на работающем сервисе. Естественно эксплуатация «в шоке».
  Что с этим делать. Во-первых, уведомить отдел эксплуатации, во-вторых – усилить команду поддержки тем же «девопсом».

О чём знает служба эксплуатации и могут не знать все остальные:

- **Пользовательский опыт по работе с продуктом**. При DevOps подходе – главный KPI это time to market. Ключевое слово «market» – рынок, что означает, что продукт продаётся, а значит присутствуют такие понятия как удобство, решение проблем пользователя, простота.
  Иногда «удобство, простота» просто не учитываются разработчиками, а «решение проблем пользователя» вообще ни о чём не говорит. «Мне дали задачу, я её сделал». То, что решение одной проблемы пользователя может привести к раздражению пользователя и создании у него трёх новых проблем, никто не думает.
  Что с этим делать. Хорошей практикой сейчас есть дежурство отдельных разработчиков в поддержке, пусть даже в режиме наблюдателя. Иногда, а лучше все время, разработчики должны смотреть на свой продукт со стороны пользователя, со стороны бизнеса и со стороны эксплуатации. Сразу открывается много новых горизонтов по оценке продукта, своей роли, своей эффективности. Естественно, это все справедливо, если у команды есть определённая система мотивации делать качественный продукт и развивать его.
- **Проблемы в инфраструктуре.** Всегда что-то будет происходить. Нет полностью «спокойного» мониторинга. Есть нагрузки, узкие места и прочие атрибуты высоконагруженных систем. Разработчики об этом не знают, да и не особо должны знать. Их дело написание кода и рефакторинг. Но нехорошие моменты бывают.
  Что с этим делать. Важно знать всю картину на момент высокой загрузки или сбоя. Должны подключаться все службы и решать проблему – рефакторингом, усилением безопасности, деньгами (в случае недостаточных мощностей) должны все. После сбоев всегда происходит «разбор полётов» – процесс post mortem, в котором фиксируются абсолютно все события инцидента. Есть даже специальное ПО для такого типа задач – Morgue.
- **Инциденты и сторонняя активность**. DDoS-атаки, возросшая нагрузка на API и бэкенд, инфраструктурные сбои хостинг-провайдеров, сбои в сервисах подрядчиков. Всё это неотъемлемая часть любого сервиса. Это надо принять и быть к этому готовыми. Первыми о таких событиях узнает эксплуатация.
  Что с этим делать. Следующая информация должна быть «под рукой» у всей команды:
  - Чек-лист по зонам ответственности сервиса. Появляется после изучения архитектуры сервиса.
  - Список подрядчиков сервиса и список контактных лиц, с которыми можно решать вопросы. Все должны знать на каких моментах какие контакты подключать.
  - Список контактов и полномочий своей компании. Кто за что отвечает, кого в какую очередь подключать.
  - Полная информация по всем параметрам инфраструктуры. Учётные записи в панели управления доменами, личные кабинеты и т.д.
  - Знать алгоритм получения логинов/паролей к любому объекту в инфраструктуре сервиса

Также надо выделить отдельную роль «летописца», который будет фиксировать все этапы инцидента и как с ним «справлялись» все службы.

Сейчас даже есть прекрасный тренд выставлять в публичный доступ описание инцидентов и как компания с этим справлялась.

> Как ни странно – это очень повышает доверие и лояльность, когда компания честно рассказывает своим клиентам и акционерам что произошло и как выходили из нехорошей ситуации.

Пару слов о чек-листах.

С точки зрения создания, продумывания, тестирования хороший чек-лист – это произведение искусства. Это даже не документация – написал, посмотрел, что работает, отдал дальше… а там уже все зависит от «потребителя», как он понял, на сколько

он владеет предметом, как он это будет использовать. В любом случае – большая часть ответственности с автора снята.

Чек-лист – это совсем другой принцип. Это очень краткое руководство к действию, которое не снимает ответственности с автора. Потребитель чек-листа не должен понимать все процессы от А до Я, он просто должен сделать и отметить все пункты. Наше мнения в этой ситуации разделяются.

С одной стороны, чек-лист позволяет легко масштабировать службу поддержки или эксплуатации. Есть действия, которые надо пройти или выполнить и, если «не работает» – эскалировать заявку «старшим товарищам», пусть они разбираются дальше, мы заявку на своём уровне «закрыли».

С другой стороны – а зачем тогда нужна поддержка, которая не понимает, что она делает?

Если есть желание автоматизировать работу с чек-листами и внедрить у себя в компании – есть прекрасный бесплатный плагин для Jira – Structure.Testy.

Правила эскалации заявок

Следующий момент – правила эскалации. Разработчики должны понять, что их в покое никто не оставит.

Чтобы operations получал актуальный чек-лист и инструкции необходим отдельный отдел технических писателей, которые обновляют документацию.

Это при условии, что в отделе поддержки нет сильной загрузки, текучки, люди достаточно «дорогие» и опытные, нет саботажа по закрытию/открытию и эскалации заявок. Другими словами, поддержка не «перебрасывает» через забор заявки, с которыми она не хочет/не может справляться.

Ситуация на самом деле довольно скользкая. Поскольку DevOps это некий мост между разработкой и поддержкой, то самый простой вариант эскалации заявок на разработчиков через него. И на первых порах возможно так и придётся делать.

Техподдержка (то, чего они не поняли/не хотят понимать) > DevOps (оценка зоны ответственности, самостоятельное решение) > Разработка (то, что не удалось решить DevOps)

DevOps и бизнес

Правильней начать с целей найма «девопса». Какие они могут быть с нашей точки зрения:

- **Ускорение вывода продукта на рынок.**
  Time to market. Основная цель и основа всех DevOps-трансформаций. Отработка идей маркетологов в усиленных темпах, быстрое создание конкурентных преимуществ.
  Ускорение onboard новых технических сотрудников.
  Два месяца на «раскачку» по среднерыночной цене разработчика от 2000 у.е. в месяц или две недели?
  Построение прозрачных процессов разработки.
  Все видно в Jira, Slack, ELK. Работающие бизнес-метрики, понятный процесс работы с инцидентами.
  Грамотный мониторинг цен и управление стоимостью сервиса.
  Почему такая цена инфраструктуры, как можно сэкономить? И заход с другой стороны – сколько денег нам надо чтобы мы могли «переварить» «чёрную пятницу», если учесть, что рекламный бюджет увеличен на 200% по сравнению с прошлым годом?

Ускорение вывода продукта на рынок

> Классический вариант «взаимодействия» без подробных обсуждений и торга:

1. «Продажи и маркетинг» перекидывают требования через стену, за которой сидят «программисты»
2. Через какое-то время «программисты» перекидывают обратно «изделие», которое сделали согласно своему пониманию требований из пункта №1
3. «Продажи и маркетинг» перекидывают замечания и уточнения
4. ……
5. ……
6. ……
7. Сроки по основному проекту сорваны, «изделие» для «продаж» не работает. Все собираются на ретроспективу и жалуются друг на друга руководству.

Один из вариантов – усадить «Продажи и маркетинг» и «Программистов» за стол как клиента и команду. Пусть не будет настоящего бюджета – хоть покерные фишки всем сторонам раздайте. Пусть результат такого мини-проекта, будет конвертирован в премии или выходные. Но все стороны придут к какому-то решению и все будет идти по нормальным процессам: техзадание, оценки сроков, приоритеты, риски по внедрению и возможной деградации основного сервиса или продукта.

В итоге «продажи» должны понять, что разработка их пожеланий – это часто достаточно трудоёмкий процесс, который иногда требует переделки архитектуры.

Например, есть какой-то сервис, у которого есть кабинет пользователя. «Продажи и маркетинг» решают, что неплохо было бы в кабинет добавить чат с личным менеджером. Они такое видели у конкурентов или у других сервисов. Удобно, НАВЕРНОЕ повышает лояльность клиентов.

Создаётся задача – «Нам нужен чат для клиентов в личном кабинете со своим менеджером, когда сделаете»?

Что происходит «за стеной»:

1. 1. Оценка задачи и уточнения.
   Чат это web-socket соединение, что значит соответствующий сервер. Продумать какие таймауты, как шифровать. Посмотреть на готовые решения. Или написать свой модуль для чата. Вёрстка, дизайн – передать на утверждение. Функционал чата – нужны вставка картинок, вложение файлов, форматирование текста, история переписки? Нужна поддержка голоса или нет? Где держать вложения, нужен ли к ним доступ «своего менеджера»? Что такое «свой менеджер»? Где брать о нем информацию и как связывать с клиентами. Если он берётся из CRM, значит надо разрабатывать связь с сервиса с ней.
2. Эскалация на Ops и DevOps – нам нужно новое окружение для этого проекта. Ops или DevOps идёт к бизнесу за деньгами.
3. Список вопросов для уточнения у «Продажи и маркетинг».

   Просмотр загрузки по разработчикам, meeting по архитектуре, оценка сроков реализации, критерии приёма задачи. Следующий этап переписка и выяснение деталей. «Продажи и маркетинг» не понимает, о чем их спрашивают. Начинается назначение ответственных со стороны «Продажи и маркетинг». Особо неопытные могут дать анонс нового функционала. Через несколько итераций в 90% случаев получаем результат: разработчики потратили время и ресурсы на обработку и имплементацию задачи, «продажи и маркетинг» не получили нужный функционал

Как можно сделать правильно (наше видение).

Создаётся дополнительный проект «Чат для клиентов». Может быть, для начала просто надо сделать «пустышку» в кабинете клиента со значком чата и отследить сколько пользователей нажмут на неё?

Можно провести исследования рынка, посмотреть есть ли у аналогичных продуктов такие функции. Можно смоделировать потенциальную прибыль от внедрения такого чата в кабинете – пользователь и так зашёл в кабинет, значит сервис ему интересен, и он им пользуется. Что даст чат в плане продаж? Через сколько времени затраты окупятся прямо или косвенно? Как это отследить и посчитать?

Со стороны «Продажи и маркетинг» назначается Product Owner, который формулирует требования.

ПМ со стороны разработчиков выясняет функционал, его важность и приоритетность. Создаётся ТЗ (подключается технический писатель). Рассчитываются сроки и ресурсы. Считается цена разработки в «покерных фишках».

«Продажи и маркетинг» смотрят на окупаемость проекта «Чат для клиентов». Если не уверены в успехе – перечисляют какое-то количество «покерных фишек» разработчикам за разработку ТЗ и услуги технического писателя. Которые могут быть сконвертированы в выходные для команды, например.

Надо понять, что сейчас команда – это и так дорогой ресурс. DevOps с помощью автоматизации и отлаженных процессов может сделать разработку быстрее и эффективней, но дороже – оплата DevOps трансформации, работы специалиста, мощности для автоматизации. Проверять слабые маркетинговые теории ради возможности самой проверки очень затратный путь. Хотя вот они, программисты, в соседней комнате сидят.

> Все должно проверяться и тестироваться – теории, технологии, процессы. В современном процессе разработки нет «серебряных пуль». Но всегда есть «расплата» за прогресс.

Хотите простые микросервисы – получите коренную переделку продукта (привлечение консультантов, обучение разработчиков), API на все случаи жизни, документирование API, мониторинг связей между сервисами, APM-мониторинг.

Хотите Docker – получите оркестрации, игры с хранилищами и сетью.

Хотите AWS – получите vendor lock, цену, привлечение консультантов, обучение разработчиков (AWS Lambda и т.д.). Ну и возможные счета за оплату.

«Давайте внедрим Scrum» – кто product owner и, кто будет scrum-мастером? Оказывается, надо и со своей стороны учиться для внедрения Scrum.

Про «внедрить DevOps» написано в книге. Всегда надо подходить с позиции тестирования и проекций на текущую ситуацию с командой и продуктом. Делать ради «быть в гущи событий» интересно, но начинать надо с каких-то фокус групп.

Построение прозрачных процессов разработки

Любые процессы работают только тогда, когда ими все пользуются правильно и в полной мере. Есть правила игры, их все поняли и приняли. Для этого и вводятся методологии разработки.

Сформировано чёткое построение – фаланга или легион, не важно. Есть строй бойцов, которые знают, что и когда им делать. «Ломать» строй никогда нельзя. Нельзя в построении быть привилегированной особой и ходить как вздумается. Даже ферзь в шахматах может ходить только в определённых направлениях.

Agile или его разновидность Scrum это прежде всего процесс. Он мало формализован, но какой-то порядок есть. И главный порядок – это спринты.

Это обсуждённый, и самое главное, принятый всеми сторонами порядок работ.

Задачи спринта должны быть константой на срок спринта. Все. Других правил нет. Этот процесс не идеален, но лучше он, чем анархия. Если анархией занимается командир, то ждите анархии от солдат.

Мы уже не говорим, что поломать спринт – это дикая демотивация. Не в плане, что с вами будут не согласны.

А в плане того, что через какое-то время **РАЗРАБОТЧИКИ НЕ БУДУТ ВИДЕТЬ СМЫСЛА В ПЛАНИРОВАНИИ**. Для вас, как для «центуриона» фаланги бизнеса – это будет означать, что ваш строй (легион, фаланга) разрушен. Через какое-то время, это будет «толпа варваров» где каждый сам за себя.

Какой смысл тратить время на «нарезку» задач, оценку сроков, когда они через три дня будут другими??? Лучше сидеть и «ждать таску в джире».

Причины «поломки спринтов» могут быть разными, может даже и объективными,

но порядка уже не будет. Будет выгорание разработчиков, бунт и увольнения.

Всё это может перекликаться с ситуациями, описанными в пункте взаимодействия маркетинга и разработки.

Метрики DevOps

Как оценить работу DevOps? С точки зрения бизнеса хочется все описывать фразой «Я хочу, чтобы всё работало». В 99% случаев в ИТ все работает. Все “пингуется”, пакеты проходят. Только пользователи сервиса уходят к конкурентам.

Постепенно начинают придумывать метрики измерения результативности процессов DevOps. Наиболее обговариваемые сейчас:

- **Lead Time** – время между написанием кода и доставкой в production.
- **Deployment Frequency** – частота “деплоев”.
- **Mean-Time-To-Recover (MTTR)** – как быстро команда восстановит сервис после деградации.
- **Change Fail Rate** – какой процент развёртывания приводит к ухудшению качества обслуживания или перебоям в работе системы.

По нашему мнению, это чистая «синтетика», но просто надо с чего-то начать, с каких-то «попугаев».

Практическое значение этих показателей очень сомнительное. Сильно поможет прибыльности сервиса рост «деплоев» кода с регрессиями 10 до 50 в день?

Почему решили, что именно развёртывание в метрике Change Fail Rate приводит к деградации сервиса?

Lead Time – тесты не прошли, код не ушёл в production, кто виноват?

Постепенно начинает приходить понимание, что метрики эффективности DevOps это больше про всю команду. И польза есть чаще косвенная. И измерить её достаточно тяжело

.

Внедрили автоматизацию тестирования или DevSecOps – Lead Time катастрофически увеличилось, тесты не проходят, приходится переписывать. Зато качество кода улучшилось, что даст уменьшение деградации сервиса в будущем, наверное. За что ругать и за что хвалить в такой ситуации?

Есть возможность «обсчитать девопса» для зарплатной ведомости, по более техническим параметрам, которые пришли из ITSM, утвердить SLA, рассчитать основные тайминги, за которые он может сделать типовые задачи: поднять окружение, написать docker-файл.

Но это больше в упор на Ops. И опять «попугаи».

> Самый реальный способ контроля и измерения эффективности – сделать отдельный проект для DevOps с поэтапным внедрением «фишек», которые описаны в книге.

Но и тут есть подводные камни. Считать выполненные задачи можно, но это достаточно «близорукое» решение – DevOps внедряется для эффективности процесса разработки, а не для «закрытия тасок».

Ну поставит DevOps в «Done» в Epic «Внедрить в команде DocOps». А команда до использования этой опции или не дошла, или охладела в процессе, а может просто не хотела никакого DocOps вообще. И писать документацию не хочет и не будет, вернее будет, но потом.

Поэтому надо ещё «не берегу» садиться, продумывать и проговаривать метрики, от которых будет зависеть сотрудничество с DevOps.

Технический долг по документации

Документация пишется всегда тяжело. Это не совсем творческая работа.

Очень многое во время работы делается на полуавтомате и надолго не задерживается в памяти. Если что-то делается вручную, то 100% будут ошибки, опечатки или что-то забудется. Полностью все описать нельзя.

Со временем по документации будет накапливаться технический долг. Он будет расти благодаря двум факторам:

- запросы от пользователей документации: ошибки, просьбы обновить устаревшие данные, описать более подробно какие-то моменты
- самостоятельное развитие документации и CMDB: окружения постоянно меняются и создаются новые, продукт или сервис постоянно развивается

Соответственно необходимо время для «закрытия» этих долгов. Надо помнить несколько правил:

- если документация устарела, то считайте, что её нет
- если документации по какой-то объёмной теме нет, то новые сотрудники будут все писать или настаивать с нуля вместо того, чтобы разобраться.

Для бизнеса это означает выброшенные деньги.

Грамотное управление стоимостью сервиса

Как правило, DevOps управляет мощностями. А это всегда деньги. Предвидеть количество необходимых мощностей достаточно сложно – начиная от изменения каких-то архитектурных решений и заканчивая ошибками в реализации, которые могут вызвать сильный рост потребления траффика или начать писать много log-ов.

Иногда увеличение мощности окружения избавляет разработчиков от оптимизации кода и даёт быстрый результат. Но это не очень хорошая практика.

Мы написали отдельный раздел, в котором расписано на что DevOps будет просить деньги. Тут даже вопрос не в деньгах, а в скорости и сложности их получения, или оплаты необходимого. Бывают ситуации, что надо очень быстро, ну вот моментально. Надо чтобы бизнес был как-то к этому готов.

Ошибки при внедрении DevOps

На волне хайпа всегда можно потерять время и деньги. Можно совершать ошибки, думая, что ты экспериментируешь. Но попытка вставить круглое в квадратное – это не эксперимент, а глупость. С учётом текущих зарплат на рынке «эксперимент» может вылиться в круглую сумму.

Ниже мы привели основные ошибки, которые с нашей точки зрения совершает большинство организаций, пытающихся внедрить у себя процессы DevOps.

DevOps – это не универсальная «серебряная пуля»

«Серебряной пулей» всегда можно выстрелить себе в ногу. Вы ровным счётом ничего не улучшите, если само улучшение вам принципиально не надо. Все описанные в этом документе процессы и инструменты – это просто обобщённая практика.

Все это было уже раньше в виде скриптов и т.д. Просто никто это не называл какими-то терминами и не устраивал ажиотаж.

> Может у вас уже есть bash-скрипт, который собирает проект и выкладыва-

> ет его на сервер. Только у вас это называется «скрипт, который собирает и выкладывает на сервер», а не CI/CD. И Jenkins вам для этого не нужен.

Бывают ситуации, когда главный конёк DevOps – автоматизация и унификация вообще смертелен. Например, когда компания работает в сфере b2b, имеет коробочный продукт и у её клиентов разные окружения и разные версии продукта. И при этом критичен простой приложения.

Ни о какой автоматической доставки артефактов не может быть и речи. Это правда не означает, что надо отменять непрерывную интеграцию или автоматическое тестирование.

Трансформация происходит не у всех

Любое внедрение процессов – это трансформация или разрушение старых. Меняться должны все, менять подходы, отношение и вовлеченность.

Что толку от автоматизации создания документации, если её никто не читает? Какой смысл в непрерывной интеграции, если сборки все равно ломаются и тесты не проходят? Нет смысла в изменениях если изменения никому не нужны или нужны только руководству.

Переименование «лишних» разработчиков

> «У нас есть Сергей, он хороший разработчик, пишет на PHP и знает CentOS. Сейчас для него нет задач, давайте сделаем его DevOps-ом».

Нет должности DevOps. Никто же не говорит сделать Сергея Agile-инженером или просто Agile.

DevOps это методология и в какой-то мере инструменты. Если «условный Сергей» сможет описать своими словами все инструменты и процессы, которые описаны в этом документе, владеет описанными инструментами и готов нести ответственность за результат, то да – «делайте Сергея DevOps-ом», они сейчас очень нужны.

Неправильные цели DevOps трансформации

> «Нам надо внедрить DevOps и настроить CI/CD». В переводе обычно это означает настроить GitLab, Kubernetes. Это не внедрить DevOps.

Внедрение DevOps должно служить только одной цели – развязать руки «бизнесу» для отработки новых идей продукта, создания конкурентных перимуществ.

Поэтому и нужна эта дикая автоматизация всего и вся, эти окружения, которые создаются по любому щелчку пальцами – максимально разгрузить разработку от непрофильных задач и освободившиеся ресурсы пустить на проверку теорий по улучшению продукта маркетологов или владельцев бизнеса.

Почти никто не работает с документацией

Все знают того, кто всё знает. И в случае чего у него спрашивают.

Даже если кого-то и заставили написать хоть пару страниц, то про обновление можно забыть.

Этот процесс системный и скорее всего уже необратим – по всей видимости усложнение продуктов и темп разработки уже набрали такую скорость, что вести документацию просто не выгодно. Усилия по её поддержке не окупают результат.

Но при этом все разработчики хотят присоединяться к хорошо описанным проектам

и тратить мало времени на разбор кода и архитектуры. По всей видимости должность технического писателя будет уже обязательной для ведения внутренней документации проекта.

DevOps не архитектор

Он должен понимать, как работает проект или сервис. Он должен, в итоге, это как-то описать. Но он не может нести ответственность за архитектуру, поскольку он не имеет влияния на разработчиков, почти не участвует в разработке, у него другие задачи и роль в иерархии проекта.

Для таких вещей есть системный архитектор. DevOps ответственен за проект с точки зрения мощностей, отказоустойчивости или балансировки. На нем будет ответственность за корректный доставку и развертывание кода на production. Он будет решать (или оказывать влияние на решение) какую из схем при этом применять.

DevOps не support

Он обязан по максимуму обеспечить поддержку информацией о сервисе, инструментами для его контроля. В идеале он должен или разработать workflow поддержки, или принимать в этом самое активное участие.

Естественно на него должна быть эскалация по каким-то важным вопросам, но объединять роли support и DevOps в одном лице – это большая ошибка. Такой подход может быть оправдан при создании службы поддержки, но делать это системно нельзя.

DevOps не может знать все

Спектр технически решений, которые должны знать DevOps достаточно широк.

Но есть определённые области где не справится даже команда из сильных специалистов. Речь идёт о anti-DDoS, DNS-балансировке, WAF (Web Application Firewall).

Все эти решения защищают разрабатываемые и сопровождаемые сервисы и должны применяться по умолчанию. Поэтому, если сервис имеет много пользователей или готовится к расширению – однозначно арендуются соответствующие услуги у Cloud Flare, Amazon и т.д.

> Причет не бесплатные тарифные планы. Никаких вариантов из серий «я слышал с помощью haproxy можно отбивать DDoS» или «а если просто прикрутить к Nginx антивирус какой-нибудь нам это поможет?»

Не надо переносить все в облако

За Amazon будущее. Это уже все поняли, пережили, осознали. Дорого, удобно, надёжно.

Понятно, что там должны быть все окружения, бэкапы, управляться через Terraform. Но, например, сервера для сборки, мониторинга и хранения log-ов могут быть совершенно произвольными.

Хоть купить б.у. и поставить в дата-центре. Причём они будут так же управляться по методике Infrastructure as code (IaC) и «девопсовость» от этого не ухудшится. Экономия будет значительная. Лучше на эти средства купить более дорогой пакет у Cloud Flare.

Несколько советов «девопсам»

Пару слов к коллегам, настоящим и будущим.

Естественно мы не претендуем на истину в последней инстанции и не хотим кого-то поучать.

Просто это подборка даже не ошибок, а тех вещей, на которые не обращаешь внимания, когда сильно углублён в работу или решение каких-то сложных задач. Тем не менее, помнить об этих вещах надо и каким-то образом определиться как с ними работать и сосуществовать на проектах или в компании.

Выясняйте методики ветвления git и другие базовые основы в команде

Про методы ветвления несколько раз упоминалось в книге. Это очень важно. Изучите их, поймите разницу, поймите, что не все системы CI/CD одинаково хорошо поддерживают все методы ветвления.

Когда вы выясните каким flow пользуется команда, тогда можно искать подходящий инструмент. Веселее, когда несколько команд, каждая со своим flow, и ваша задача комбинированный CI/CD.

Не выпадайте из процессов и из жизни команды и проекта

Даже если вы на «удалёнке» или временно на проекте.

Надо понять, что вы и есть определённый «двигатель-ускоритель» проекта и соответственно бизнеса. Если вы будете вместе с остальными «ждать таску в джире», то может случиться, что «тасок» у вас скоро может и не быть, вместе с зарплатой или контрактом.

> DevOps – это процесс. Если вы не строите свои процессы, значит скоро появится человек, который будет их строить для вас и вместо вас. Причём он будет это делать, не особо вдаваясь в подробности DevOps и вашу роль.

Рассматривайте проект и свою работу с точки зрения бизнеса. От «девопсов» именно этого и ждут. Нам это необходимо в первую очередь.

Вы же видели красивые картинки с пересекающимися кругами по поводу DevOps. Никто кроме вас это «пересечение» кругов не осмыслит и не конкретизирует. Придётся вникать и в остальные «круги». Самое главное, что от нас требуется – time to market, в гордом одиночестве вы это не потянете.

Разрабатывайте и внедряйте свои метрики

Надо понять, что нормальный процесс найма технического специалиста не может быть основан на хайпе. Это может быть только временное явление. Всегда есть критерии оценки работы. DevOps пока обделён таким вниманием.

Мы сейчас готовим серию статей по метрикам – все очень расплывчато и грустно (следите за нашими публикациями). Есть определённый набор метрик для dev-ов, тестировщиков, ops-ов. Они вполне рабочие и справедливые. Для нас пока нет. И это опасно, поскольку нашу работу могут оценивать несколько «своеобразно».

Естественно после того, как пройдёт период «Фух… мы наконец нашли девопса не за все деньги мира (или за все деньги мира)». «Бизнес» «не дурак», и хочет понять за что в итоге он платит от 2500+ до 5000++ в месяц (плюс печеньки и спортзал).

> Метрика «чего вы придолбались – все «деплоится и работает» это не метрика. Это может сработать на протяжении «конфетно-букетного» периода. Потом надо уже показывать «было/стало».

Повторимся – DevOps это все-таки про бизнес. Поэтому надо очень активно общаться с маркетологами и финансистами и понять, как количество «деплоев» в месяц с 5 до 500 помогло увеличить прибыль.

Мы же понимаем, что бизнес разный и цели достижения прибыли тоже разные.

Может быть такое, что 500 «деплоями» вы помогли поднять продажи на 5000 долларов, а автоматизация сборки и обновления документации косвенно сэкономила 20000 долларов, поскольку новые разработчики начали делать продуктивные коммиты через 2 недели, а не через 2 месяца как обычно было раньше.

Ну или поиграться со «спотовыми» instances на Amazon. Простор для фантазии огромный. Более того – есть команда, есть ПМ, есть Product Owner, все они могут помочь вам с целями и критериями оценки.

Включайтесь в маркетинг

Обсуждайте, спрашивайте, придумывайте метрики для бизнеса. Помогайте их имплементировать, для этого есть Prometheus.

Через некоторое время маркетинг и «сейлзы» будут начинать рабочий день со своего экрана в Grafana. Предлагаете или требуйте увеличение мощностей production для А/Б тестирования. Лучше опробовать одновременно условно 5 форм регистрации и закрыть вопрос с привлечением клиентов на какое-то время, чем делать это последовательно на протяжении полугода, прожирая инвестиционные деньги.

> На одном проекте маркетинг начал делать лэндинги после того, как пустил рекламу на главную страницу сервиса, на которой по web-socket клиент получал хорошую такую пачку данных.

Отделяйте мощности для лэндинга от мощностей production. Маркетинг в большинстве случаев не будет понимать, о чём вы говорите – объясняйте, рисуйте схемы, учите их тоже. Мы же помним – time to market.

Защищайте и отстаивайте бюджеты

Вам придётся просчитывать не только dev, stage, production. Если мы говорим про полноценное внедрение DevOps процессов, то надо закладываться на:

- Continuous Integration – тестирование перед сборкой, сборка, тестирование после сборки. Куча окружений для Feature Branching с аналогичными процессами. Все прелести и возможности SonarCube. Тестирование на безопасность в рамках DevSecOps.
- Instances для автоматического нагрузочного тестирования с с помощью Selenium, Jmeter и т.д.
- Мощности для тестирования с «боевыми» базами данных – скорости обработки запросов и т.д. Значит надо чтобы где-то был или snapshot последней базы, или slave для тестов в read only режиме.
- Мощности для хранения артефактов и кеширования зависимостей
- Для DocOps – отдельный от production IP с web-server, на котором будет документация, если у сервиса большое количество пользователей
- Сборка проектов. Вы читали выше про стоимость агентов для коммерческих продуктов, вы понимаете сколько надо ресурсов для частых параллельных сборок.
- Облачные провайдеры – коммерческие пакеты и средства для выкупа Spot Instances. Да, мы хотим не изобретать велосипед, а создавать правила для WAF сразу на Cloud Flare. И это нормально и правильно.
- Коммерческие контракты на поддержку критически важных элементов инфра-

структуры. Упавший Kubernetes можно и не поднять после обновления, и видео из Highload в этом не помогут.
- Не забываем, что мы тоже творческие люди и тоже развиваемся. И нам тоже нужны тестовые окружения для проверки наших теорий по новой архитектуре. Или для тестов каких-то новых продуктов, которые мы хотим использовать в проекте.
- Мощности для мониторинга и log-ов
- Бэкапы и тестовые окружения для Jira, GitLab, etc. В среднем, раз в месяц нам приходит письмо от Atlassian с предложением вот прямо просто срочно обновиться, поскольку у них уязвимость в каком-то продукте. Ну и обновляться надо на последние версии тоже, там добавляется много интересного. А вот теперь представим, что не работает GitLabCI или Bamboo – неудачно обновились, или версия java не та. А у нас спринт. Так что начинаем быть сапожником с сапогами – blue-green deploy и вперёд. Естественно не надо вываливать сразу весь этот зоопарк запросов на голову «бизнесу». Но вы должны быть подготовлены по лицензиям, вариантам стоимости, срокам, когда это все хозяйство понадобится, этапам и очерёдности.

Чтобы у «бизнеса» не было инфаркта, не надо это все считать на основе clouds (AWS, Azure, GCP). Кучу всего из этого списка вы можете разнести по разным площадкам, или выбрать какую-то одну по соотношению цена/качество.

Да, надо промониторить кучу хостинг-провайдеров, поспрашивать у коллег, сравнить варианты. Это тоже работа и достаточно кропотливая, отнимающая время.

В любом случае, при любой ситуации, вы должны быть готовы ответить по вопросу разовой стоимости и сколько платить в месяц/год. Это мы тоже кстати про разговоры о метриках, и о том, что надо общаться со всеми. Вот и будет повод.

Не «тяните» все сами

Вы видели какой объем работ необходимо проделать для качественного внедрения принципов DevOps. Когда сроки поджимают, могут понадобится ресурсы на поиск и привлечение коллег по цеху с определёнными компетенциями, например, мониторинг и тестирование. Может технического писателя для внутренней документации. При правильной и разумной аргументации бизнес на это пойдёт.

> Поймите, простую вещь – если будет угроза срывов сроков или другие неприятности, то команда может получить от перепуганного бизнеса «ещё двух толковых ПМ-ов» (с), оно вам надо?

Учите разработчиков

Сейчас эра узкой специализации. Платформы и фреймворки развиваются настолько стремительно, что ops-ы со своими Docker, Kubernetes тихо отдыхают где-то в сторонке.

В большинстве своём разработчики достаточно поверхностно знают про базовые понятия – DNS, (S)DNAT и т.д. Это им просто не надо. Но на этом построен production.

Почаще объясняйте им как устроена инфраструктура, про все возможности балансировщиков. Например, они должны знать про «липкие сессии» (sticky sessions), которые использует ваш load balancer – это избавит их от написания подобного функционала в своих решениях по работе с пользовательскими сессиями.

Расскажите им про кеширование в DNS, что можно подключать соответствующие библиотеки, найдите им эти библиотеки. Вы облегчите себе и support работу в будущем.

Учитесь у них. Они лучше всех знают, как работает node.js, что ей надо по мощностям, почему много ядер CPU не обязательно.

Помогайте им с DocOps, с подключением APM-мониторинга, пишите инструкции как это делать. Научите их расшифровывать мониторинг, что означают все эти графики, чтобы они могли вместе с вами отлавливать баги после себя.

Всегда проверяйте возможность восстановления баз с production

А лучше делать это автоматически. Пусть вас не пугает объем. Потеря репутации компании от потери данных пользователей намного больше, чем стоимость аренды второго окружения для баз данных. Естественно это надо говорить не после потери данных (тьфу три раза) в разрезе «а я говорил», а нормально объяснить «бизнесу» зачем и почему. Можно использовать слова «сроки восстановления сервиса», «отток клиентов», «потеря репутации». Про отстаивание бюджетов написано выше.

> Хотя сейчас набирает популярности идея, что «настоящего» stage не существует. В разрезе фактического объема данных. Он на столько быстро растёт, что цена stage, который должен по всем «канонам» держать фактическую копию, выходит космической. Все вздохнули с облегчением и полезли «дебажить» production по ssh. Все как раньше.

Внедряйте и поддерживайте CMDB

Вы избавите себя от кучи вопросов по окружениям, «А сколько у нас памяти на сервере баз данных?», «Как туда попасть? Мне надо…» и т.д. А они всегда будут возникать в процессе проекта от самых разных людей.

Естественно данные должны быть актуальны, обновляться в автоматическом или полуавтоматическом режиме, остальная команда должна знать об существовании CMDB и доверять информации, которую они будут оттуда использовать.

Начните внедрять DocOps с себя

Документируйте все аспекты своей работы. Вносите изменения по мере изменений архитектуры, добавления окружений. Пишите инструкции и чек-листы.

Особенно важны инструкции по deploy/rollback, просмотру log-ов, поиску по log-ам, как вручную остановить/запустить что-то важное.

Выделяйте при оценке задачи и требуйте время на документацию, не допускайте технического долга по этой теме.

Когда-то вам придётся или сдавать дела, или вводить в курс дела новых сотрудников в своём отделе.

Автоматизируйте конвертацию и deploy документов, покажите остальным как это круто. Отсылайте к документации, если там это уже описано и актуально. Актуально оно должно быть в любой момент времени. Для этого, собственно DocOps и предназначен.

Анализируйте «тикеты» поддержки

Какие-то (если не все) тикеты вы будете создавать сами, когда на вас будут эскалировать заявки.

> Вы должны понимать, что количество открытых/закрытых тикетов это одна из метрик техподдержки, которая в итоге отражается на их зарплате. Соответственно можно много раз чего-то открыть/закрыть для «повышения надоев». Естественно не везде и не всегда это происходит.

Но анализ может дать информацию для размышлений.

Иногда бывает, что процесс передачи знаний поддержке или ops-ам построен плохо или вообще отсутствует:

- работают по устаревшим чек-листам, не знают структуры сервиса;

- не были предупреждены о возможном увеличении трафика (пустили массированную рекламу, например);
- нет передачи знаний внутри отдела поддержки – новичков плохо вводят в курс дела или вообще не вводят (ну поскольку вы же уже внедрили DocOps и теперь проблем с документацией по сервису нет);
- могут просто иметь плохую подготовку и ставить приоритеты для эскалации чтобы снять с себя ответственность.

Анализ «тикетов» позволит выявить узкие места и исправить всё это безобразие. Ну вы же помните совет про метрики и методы оценки работы, правда?

Боритесь за тесты и тестирование

В большинстве случаев тестов нет, особенно на начальных этапах проекта. Это сложно, дорого, все и так пишут правильный код.

Они планируются когда-то, потом они становятся техническим долгом. Потом выясняется, что беда, что «давайте поднимите, потом будем разбираться»; потом ревью кода; потом «все перепишем на GO». Потом вернёмся к начальной точке. А потом свернём проект.

Вы видели целый раздел в книге посвящённый тестированию. Изучите инструменты, помогите настроить статический анализатор кому-то из команды, получите обратную связь.

Имея статистику, тайминги и показатели вы можете оперировать ими как весомыми аргументами, а не «все пишут тесты, вы чего не умеете?». Не хотят тестирования, их дело. Вы можете «гонять обезьян» с support (см. Chaos Monkey).

Важно понять одну вещь – упрощение разработки без потери качества, это не о том, что разработчики и вы будете меньше работать. Это о том, что теперь все вместе смогут больше сделать.

И когда (если) продукт перейдёт из стадии «ой, смотрите, он головку держит стал пытаться ползать», до стадии продажи «ну хорошо, мы посмотрели, прикольно… а чем вы отличаетесь от…?», вот тогда и понадобится весь этот «чёртов маркетинг» с его идеями, конкурентными преимуществами и прочими непонятными вещами.

Вначале «ядро», потом обвесы и «плюшки». В большинстве случаев продажи и маркетинг ждут пока оживёт и заработает «ядро», а потом все прощаются и обновляют резюме.

Работайте с малыми проектами

Не перебегайте маленькими проектами или MVP (естественно за разумный прайс).

По сути вы и есть драйвер роста в некоторых случаях.

Вам тоже нужна проверка моделей взаимодействия с разработчиками, эффективности внедрений определённых решений и процессов.

В слаженной команде с утверждённым бюджетом проверка идей может вылезть боком, поскольку в большинстве случаев будет «legacy внедрение» процессов в уже сложившейся среде правил и взаимодействия.

Работая с «малышами» пусть даже на уровне разовых работ и консультаций, вы имеете прекрасную возможность «растить» команду и проект. Этот значительно увеличит вашу рыночную стоимость.

> Этот пункт ни в коем случае не намекает на «подопытность» маленьких проектов или команд. Все когда-то с чего-то начинали.

Послесловие

Естественно получился «первый блин комом»)). Все достаточно скомкано и рас-

плывчато. Но, с другой стороны, объем материала, который пришлось «перелопатить» достаточно большой.

Я надеюсь, что книга (вернее реферат) поможет начинающим «девопсам» или тем, кто хочет заняться этой дисциплиной вплотную. Понять, что это не совсем «сисадмин для программистов», как это может показаться со стороны.

По поводу описанных инструментов и технологий. Это просто обзор инструментов, про которые надо знать. Естественно у каждого рассматриваемого инструмента есть огромное количество альтернатив. И я жду и надеюсь, что в комментариях и обсуждениях получу целый «вал» обращений с различными вариантами формулировок «почему не описан _____, он же намного лучше, чем _____».

В свою защиту добавлю, что прошёлся по верхам и выбрал те, которые «на слуху».

> Тот же «старичок» Jenkins давно должен был уйти на покой (как я думал, начиная писать рукопись), но как оказалось – если хотите интеграцию с другими системами и гибкость, то изучить придётся. Может скоро и выйдет достойная замена, но работать надо уже сейчас.

Я сознательно не затрагивал архитектурные решения по балансировке нагрузки, высоконагруженным сервисам, service discovery, репликации и т.д. Это уже должно быть на уровне проектов и команд. Но планирую описать их в следующих версиях книги.

Правда она вырастет в объёме на порядок, но это даже к лучшему.

Не затрагивались продукты нового поколения в сфере DevOps – облачные сервисы, микросервисы, контейнеры и системы оркестрации. Это огромный объем для изучения и достаточно своеобразный по соотношению плюсы/минусы. С одной стороны, получаем гибкость и скорость (Docker, упаковка приложений, микросервисы), с другой – сложности в администрировании (хранилища для постоянных данных, «оверлейные» сети, «оркестрация»), мониторинге, реакции на сбои, методы по восстановлению сервисов. Это если разбираться на уровне выше чем «развернул Kubernetes на AWS».

Отдельно хочу остановиться на процессах, метриках и прочем. Тут, естественно, много «отсебятины» и каких-то попыток обобщить явления, которые в каждом конкретном случае могут вообще не вписаться в озвученные концепции. Поэтому я надеюсь на вашу обратную связь с критикой.

Все обобщения после ваших комментариев и исправлений я буду публиковать на сайте https://devops-service.pro, в блоге и отдельно в соцсетях. Поэтому добавляйтесь в контакты, подписывайтесь, критикуйте. После регистрации на ваш email могут прийти пару писем с дайджестом публикаций на сайте и соцсетях и ссылками на опросы на какую тему писать дальше. Я был бы благодарен за вашу реакцию. Ну и естественно там будет пункт «отписаться от рассылки», так что на счёт спама не волнуйтесь))

С уважением, Пустовит Андрей.

Содержание

Об авторе

Вступление

Технические процессы в DevOps

Couds vs standalone?

Работа с исходным кодом
- «Ветвление» Git
- Хранение и администрирование кода
  - GitLab
  - Bitbacket

Continuous integration и Continuous delivery
- Continuous Integration (CI)
  - Принципы Continuous Integration
- Continuous Delivery (CD)
- Инструменты CI/CD
  - Jenkins
  - Bamboo
  - GitLab CI (GitLab Runner)
  - TeamCity

Методики доставки артефактов сборки
- Релизы на основе среды исполнения.
  - Blue-green deploy.
  - Канареечные релизы и cluster immune systems
- Релиз на основе приложения

Хранение артефактов сборки
  - Nexus Repository Manager
  - JFrog Artifactory
  - GitLab Package Registry

Тестирование
- Методики тестирования
- Инструменты функционального тестирования
  - Selenium
  - Katalon
- Нагрузочное тестирование (load testing)
  - Apache JMeter
  - Chaos Monkey

Развёртывание окружений

Infrastructure as code (IaC)

- **Ansible**
- **Salt**

Мониторинг

- **Инструменты мониторинга**
  - Zabbix
  - Prometheus
- **Визуализация**
  - Grafana
- **Мониторинг производительности приложения (APM)**
  - Инструменты APM
    - NewRelic
    - Datadog
    - Atatus
    - Sentry
    - Jaeger

Логирование

- **Требования к инструментарию для агрегации log-ов**
- **Продукты для агрегации и обработки log-ов**
  - Graylog
  - ELK Stack
  - Loki

Документация

- **Инструментарий для DocOps.**
  - Языки разметки.
  - Конверторы
  - Сервера хранения документации
  - Confluence
  - Диаграммы
  - Документирование кода
  - CMDB
    - Инструменты CMDB.

DevSecOps

- **DevSecOps при проектировании**
- **Инструменты централизованного управления credentials.**
  - Hashicorp Vault
  - Keywhiz
  - Teleport
  - Secrets
- **DevSecOps и зависимости**
  - OWASP Dependency-Check

- DevSecOps и анализ кода
  - Инструменты статического анализа
    - SonarQube
    - Crucible
    - Open source проекты для статического анализа кода
  - Open source проекты для динамического анализа кода
    - OWASP ZAP
    - Arachni
  - DevSecOps для тестирования продукта или сервиса
- Процессы в DevOps
  - DevOps и разработка
    - Стандартный рабочий процесс
      - Доставка кода (deploy)
      - Технический регламент
        - Onboarding нового разработчика
        - Техподдержка разработчиков
        - Системы ветвления Git
        - Поведение при «поломках» сборки в CI
        - Работа с техническим долгом
        - Технологический стек разработки
      - Обучение
  - DevOps и эксплуатация
    - CMDB, документация и чек-листы
  - DevOps и бизнес
    - Ускорение вывода продукта на рынок
    - Построение прозрачных процессов разработки
    - Метрики DevOps
    - Технический долг по документации
    - Грамотное управление стоимостью сервиса
- Ошибки при внедрении DevOps
  - DevOps – это не универсальная «серебряная пуля»
  - Трансформация происходит не у всех
  - Переименование «лишних» разработчиков
  - Неправильные цели DevOps трансформации
  - Почти никто не работает с документацией
  - DevOps не архитектор
  - DevOps не support
  - DevOps не может знать все
  - Не надо переносить все в облако
- Несколько советов «девопсам»
  - Выясняйте методики ветвления git и другие базовые

основы в команде
Не выпадайте из процессов и из жизни команды и проекта
Разрабатывайте и внедряйте свои метрики
Включайтесь в маркетинг
Защищайте и отстаивайте бюджеты
Не «тяните» все сами
Учите разработчиков
Всегда проверяйте возможность восстановления баз с production
Внедряйте и поддерживайте CMDB
Начните внедрять DocOps с себя
Анализируйте «тикеты» поддержки
Боритесь за тесты и тестирование
Работайте с малыми проектами
Послесловие