

VCS в производственном процессе



Содержание

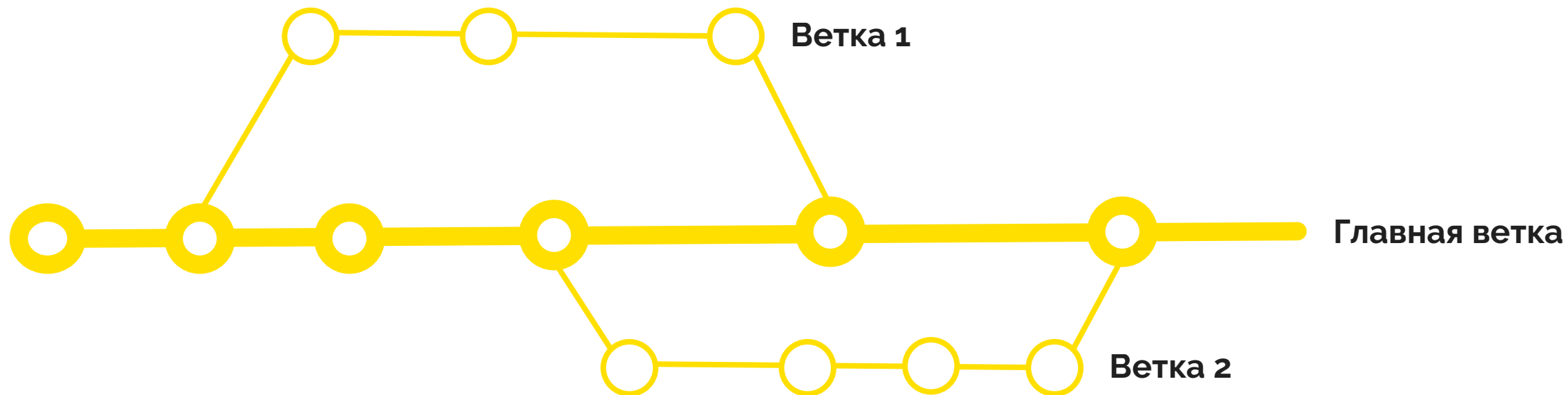
1. VCS и ее место в производственном процессе
2. Модели ветвления
3. Организация репозиториев для сложных систем
4. CI/CD

Производственный процесс



Ветки

Ветка - независимая последовательность коммитов в хронологическом порядке





Ветки

Чем помогают

1. Одновременная работа нескольких разработчиков
2. Возможность частых коммитов в собственную ветку
3. Независимая разработка и тестирование новых фич
4. Разработка нового функционала без риска сломать работающий код
5. Поддержка параллельно нескольких релизных версий ПО и возможности hot fix

Цена

1. Поддержка
2. Операция слияния веток не всегда тривиальна и порой требует значительных трудозатрат



Merge request (pull request)

MR - запрос на вливание одной ветки в другую

Зачем нужны

1. Код-ревью в команде
2. Автоматическая валидация кода перед слиянием в ветку
 - Сборка
 - Прохождение unit тестов
 - Статический анализ инструментами SAST
 - Анализ на информационную безопасность
3. Как итог – раннее обнаружение проблем и повышение качества кода

Модели ветвления

Продукты с релизными циклами (Release based)

1. GitFlow

Продукты с постоянными/частыми релизами без определенного релизного цикла

1. Github Flow
2. Trunk based development

Выбор модели ветвления – одно из ключевых стратегических решений для проекта



GitFlow

Применяется для продуктов

1. С длинными релизными циклами
2. Сложные фичи, требующие длительной разработки
3. Необходимость поддерживать несколько версий продукта

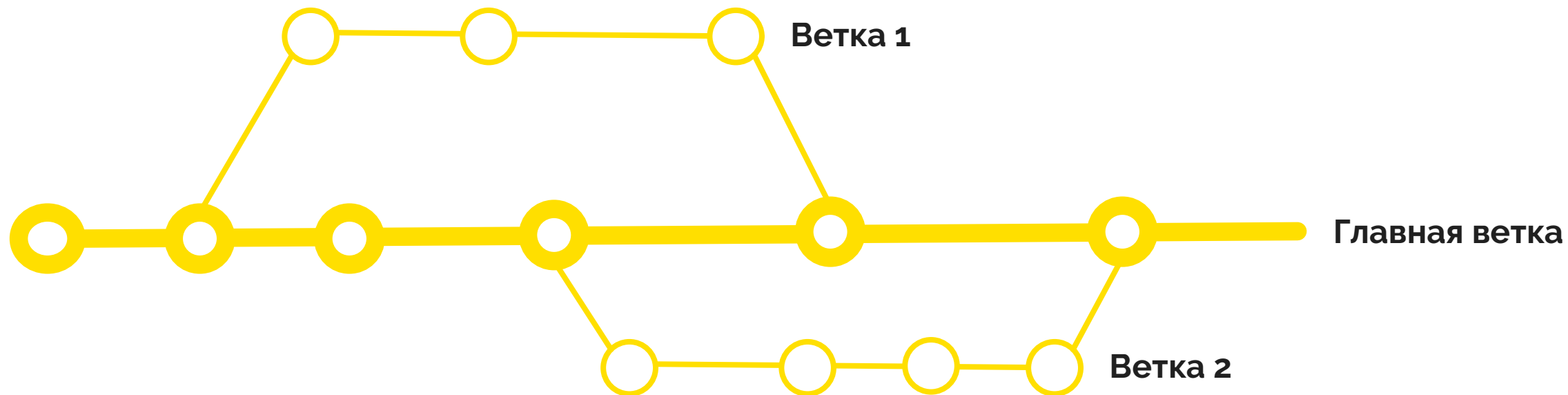


GitFlow

Основные ветки

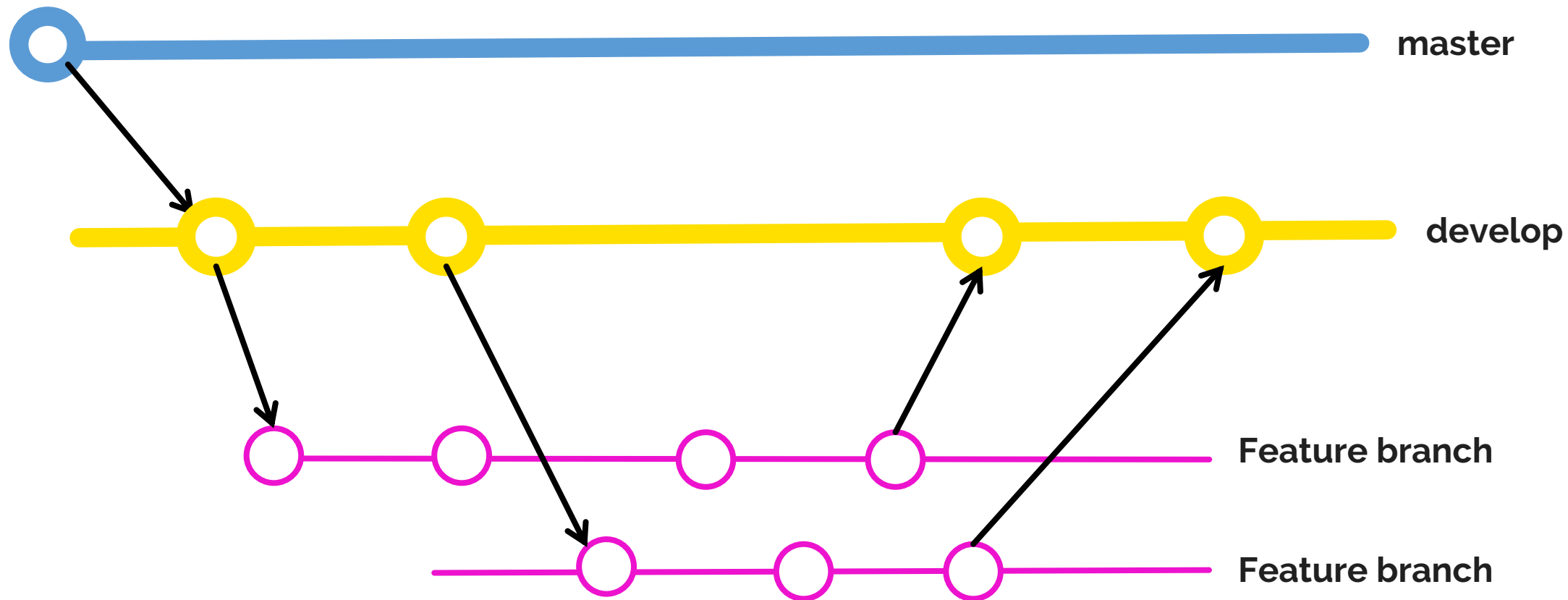
1. **master/main**: содержит тестированный, стабильный код, который может быть выложен на прод
2. **develop**: основная ветка для разработки, содержит актуальный, но не релизный код
3. **feature**: ветки для разработки новых функциональностей, которые впоследствии сливаются с develop
4. **release**: ветка для подготовки новых релизов, отделяется от develop, затем сливается с master и develop
5. **hotfix**: ветки для быстрого исправления ошибок в прод-версиях, сливаются с master и develop

GitFlow разработка и слияние

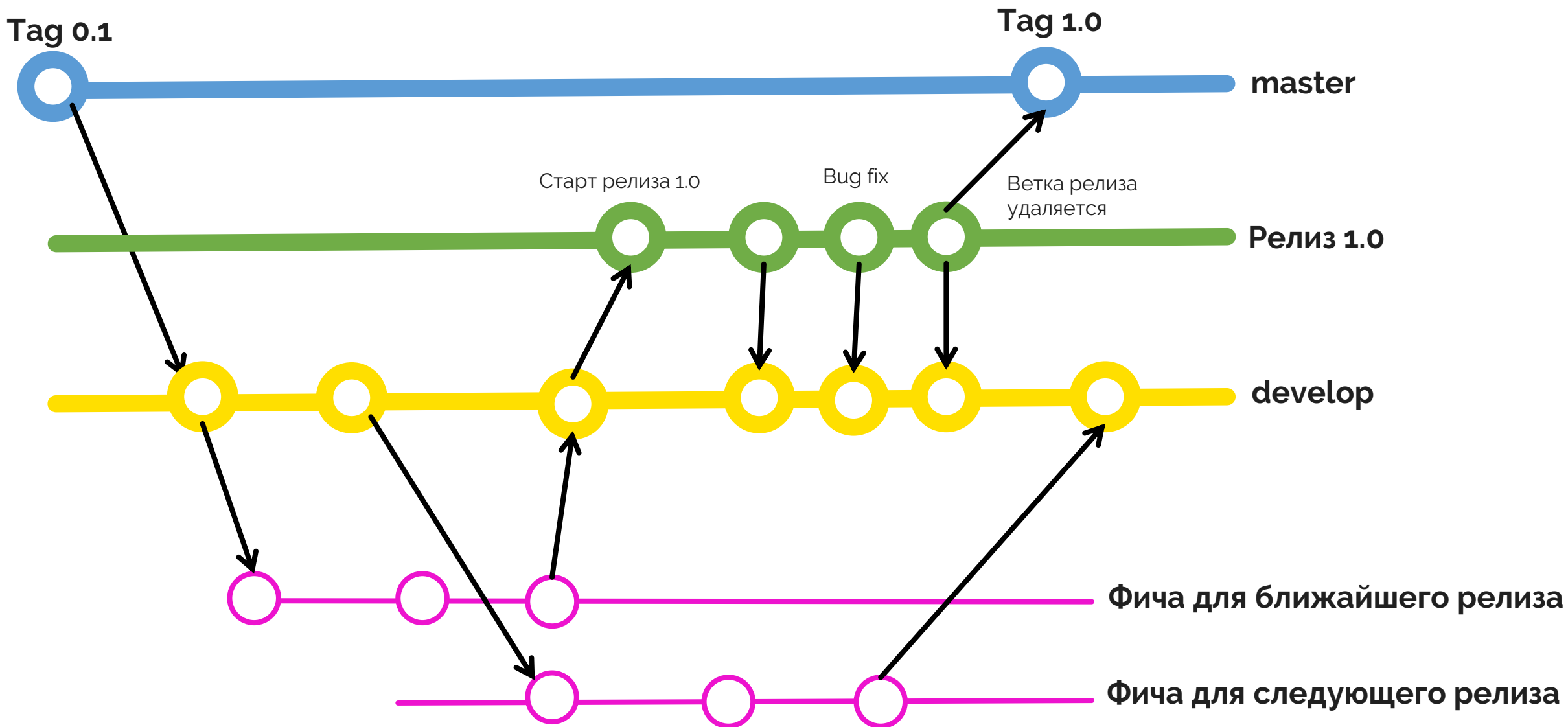


GitFlow разработка и слияние

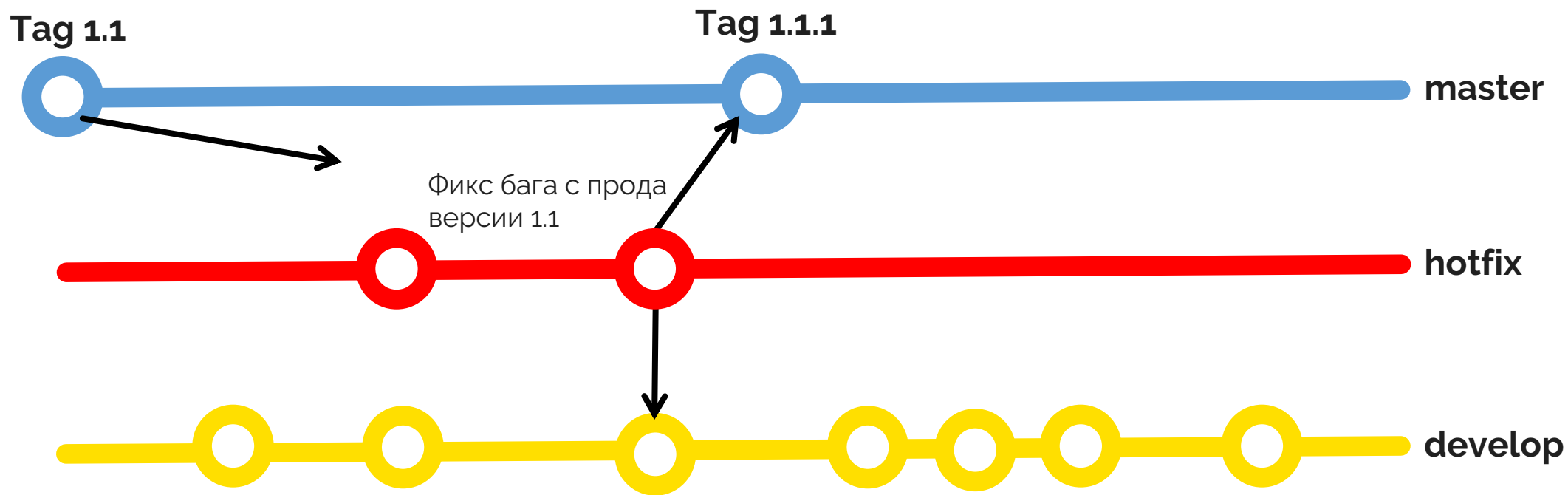
Tag 0.1



GitFlow подготовка к релизу



GitFlow hotfix





GitFlow

Плюсы

1. Четкая структура управления версиями
2. Поддержка множественных параллельных релизов
3. Удобство в навигации по истории проекта

Минусы

1. Требуется настройка правил для веток, CI/CD
2. Долгоживущие фичи. Часто ветка `develop` будет уходить вперед, поэтому при слиянии возможны конфликты



Trunk based development

Применяется для продуктов

1. С очень короткими, непрерывными релизами
2. Подходит для простых и модернизированных фич
3. Фичи должны быть независимы или использовать feature flags для управления зависимостями

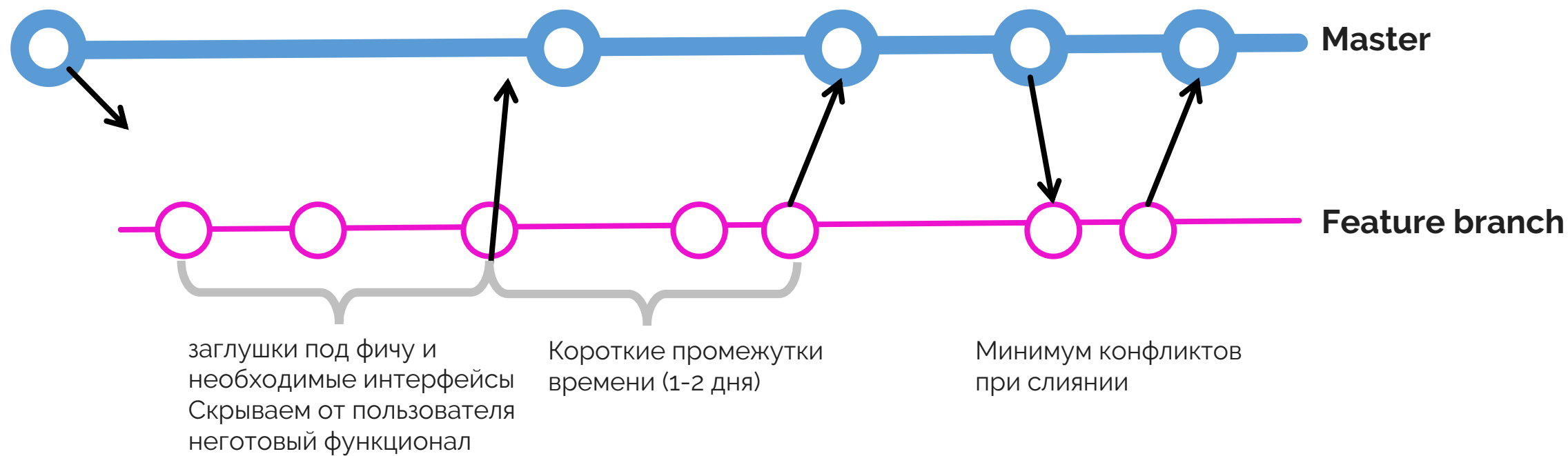


Trunk based development

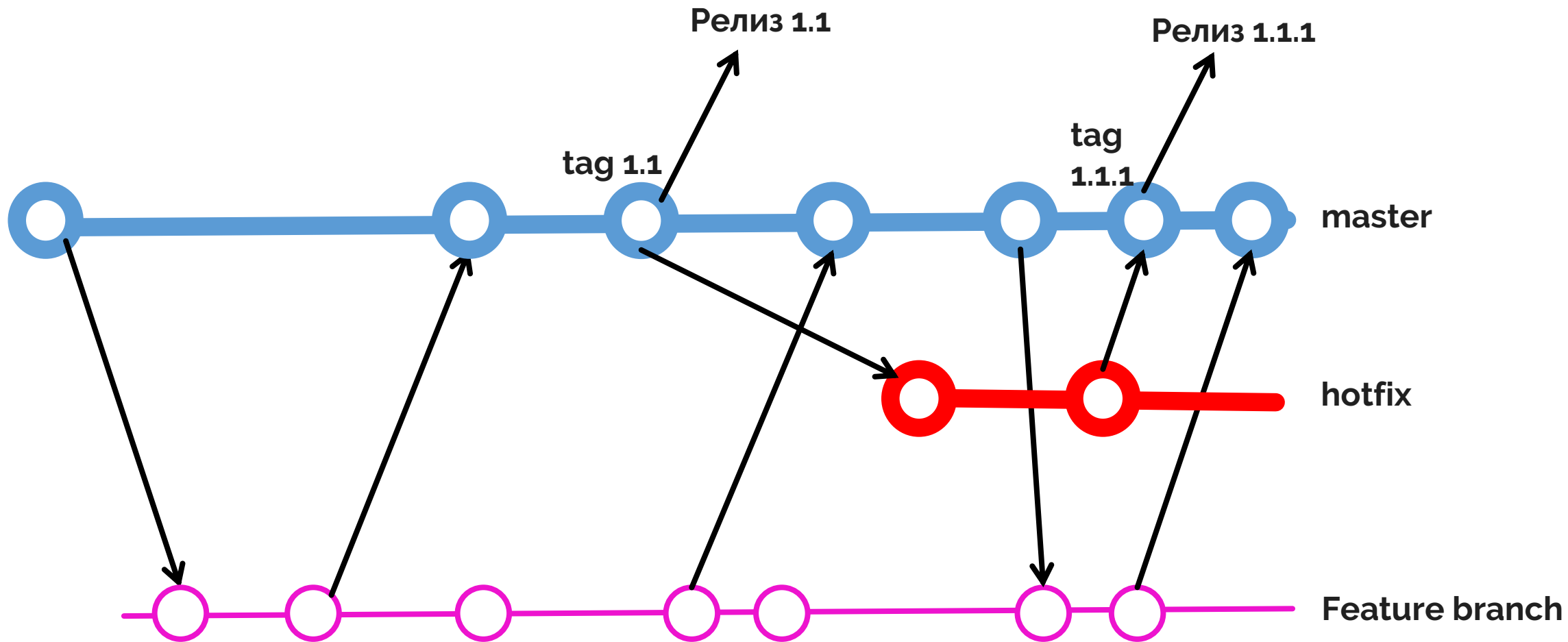
Основные ветки

1. **trunk/main/master:** центральная ветка проекта, куда вносятся все изменения
2. **feature:** в основном коротко живущие, изменения из которых быстро сливаются в master (при добавлении кода в trunk-ветку фича может быть еще не готова, а иметь реализацию в виде интерфейсов или заглушек)
3. Активное использование feature flags для контроля над функциями

TBD разработка и слияние



TBD релизы и патчи





TBD требования к тестированию и CI/CD

1. Максимально стабильный master, готовый к релизу в любой момент
2. «Сырой» функционал не должен быть доступен пользователю (использование feature flags или if false {...})
3. Подготовка чек-листов и тестирование на раннем этапе, в том числе разработчиками
4. Обязательное покрытие unit-тестами
5. Автоматизация регрессионного тестирования, встроенная в процесс CI/CD

TBD

Плюсы

1. Минимум конфликтов при разработке
2. Готовность к релизам в любой момент без подготовки
3. Легкость внедрения непрерывной CI/CD
4. Очень быстрая и качественная обратная связь на PR\MR

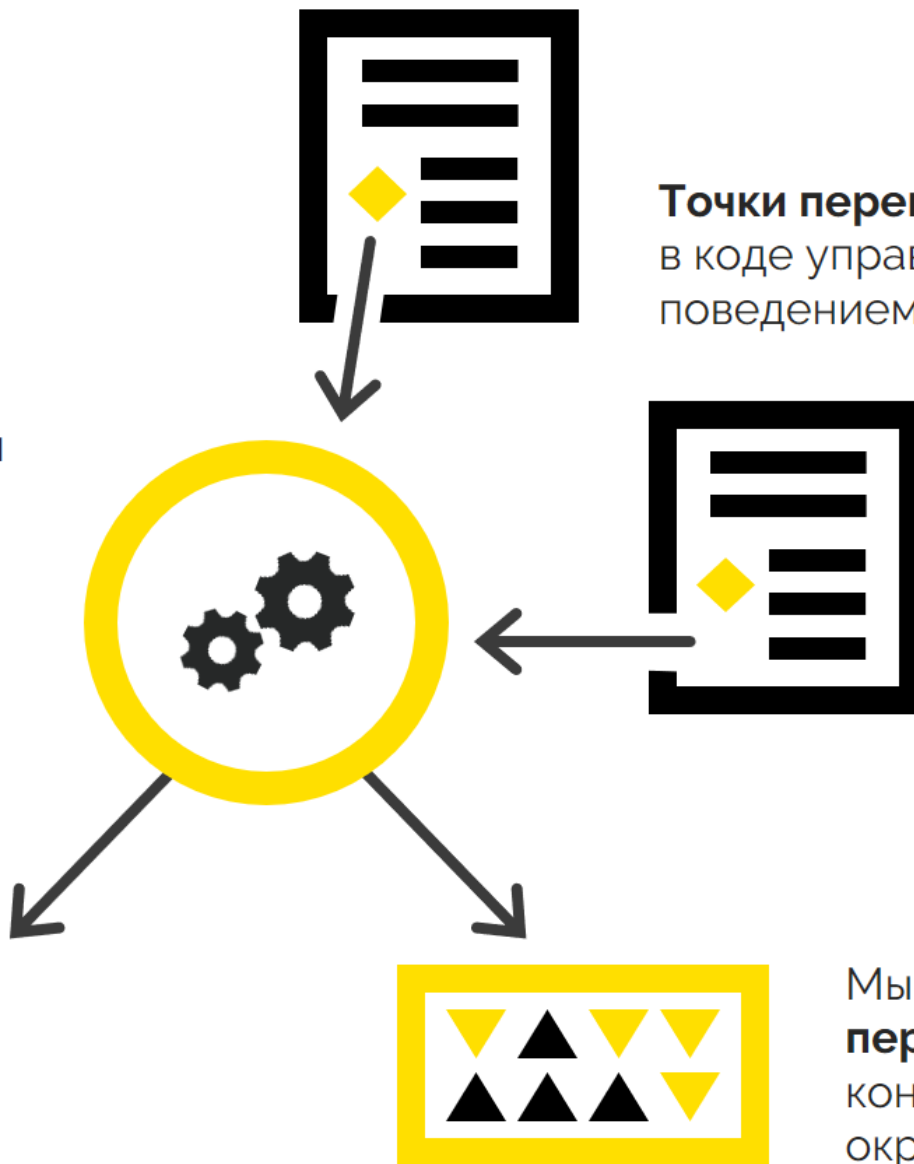
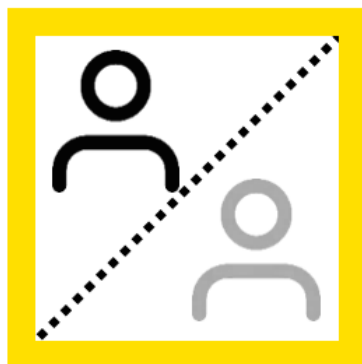
Минусы

1. Высокие требования к дисциплине разработчиков и качеству автоматических тестов
2. Риск внесения нестабильного кода в основную ветку
3. Может потребоваться дополнительное время на стабилизацию перед релизом
4. Проблема при необходимости сделать откат

Что такое feature flag?

Переключатель работает **по контексту**, например, какой пользователь отправил запрос

Одна фича, один переключатель, несколько точек переключения



Точки переключения в коде управляют поведением фичи

Мы **контролируем переключатель** с помощью конфигурации для данного окружения



Что дают Feature Flags

1. **Непрерывная доставка фич со стабильным качеством** – возможность отключить нерабочий код снижает риски в релизной версии.
2. **Тестирование новых фич в боевых условиях** – фиче-флаги позволяют постепенно внедрять сервисы, контролируя риски при релизе на реальную аудиторию.
3. **Возможность развивать несколько версий ПО параллельно** – TBD и фичефлаги позволяют предлагать разные функции разным группам пользователей, при этом поддерживать все эти версии ПО может всё та же одна команда.

Типы Feature Flags

1. **Релизные (release toggles):** скрывают неготовые фичи, уменьшают количество веток, открепляют запуск фичи от даты деплоя.
2. **Экспериментальные (experiment toggles):** используются для A/B тестирования, позволяют таргетировать функции на разные группы пользователей.
3. **Разрешающие (permission toggles):** открывают доступ к платным фичам или закрытым функциям администратора.
4. **Операционные (ops toggles):** отключают ресурсоёмкие функции. Например, так можно регулировать работу приложения на слабых смартфонах или застраховаться от падения производительности при запуске новой функциональности – флаг отключит модуль до того, как тот вызовет критический сбой.

Создание FF

Создаем до начала реализации фичи

1. Добавляем в Feature Manager новый флаг

```
public static class FeatureFlags
{
    public const string NewFeature = nameof(NewFeature);
}
```

2. Добавляем заглушки в логике всех микросервисов

```
if (await _featureManagerSnapshot.IsEnabledAsync(FeatureFlags.NewFeature))
{
    ...
}
```

featureFlags.json

Contents Highlight changes

```
1 {
2   "featureManagement": {
3     "features": [
4       {
5         "uid": "NewFeature",
6         "enable": false,
7         "description": "Флаг для новой фичи",
8         "permissions": [],
9         "customProperties": {
10          "type": "SYSTEM"
11        }
12      }
13    ]
14  }
15 }
```


Создание FF

3. Регистрируем FF на портале управления

The screenshot displays the 'Feature Flags' management interface. At the top, there's a header with 'Feature Flags 1', a toggle for 'Группировать по фичам', and a '+ Создать флаг' button. Below the header, a modal window titled 'Feature Flag' is open, containing the following fields:

- WorkItemId PBI в TFS:** 1234
- Спринт:** 25.1
- Группа:** Портал
- Задача:** Реализовать новую фичу
- Feature Flag:** NewFeature
- Тип:** Системный (dropdown menu)
- Тэг:** (empty text field)

At the bottom of the modal, there are two buttons: 'Отмена' (Cancel) and 'Сохранить' (Save).



Управление FF

1. **Проприетарные решения:**
LaunchDarkly, Bullet-Train, Unleash
2. **Open source решения:**
Moggles, Esquilo
3. **Собственная система управления**

Управление FF

1. Централизованное управление через портал ФФ
2. Отдельное состояние ФФ для каждого окружения
3. Разные стратегии включения флага

The screenshot shows a web interface for managing feature flags. At the top, there's a breadcrumb '24.6.2' and a title '#1234 Реализовать новую фичу'. On the right, there are three environment tabs: 'dev', 'uat strategy', and 'prod strategy'. Below this is a table with three columns: 'Environment', 'Feature Flipping Strategy', and a toggle control. The table has three rows for 'dev', 'uat', and 'prod' environments. Each row has a yellow callout box explaining the strategy. The 'uat' row also has a yellow callout box explaining the toggle state.

Environment	Feature Flipping Strategy	
dev	—	Включить для всех
uat	User Context Strategy login:v.pupkin	Включить для определенных пользователей
prod	Release Date Strategy Release date:01.03.2025	Включить в определенную дату

Включаем в нужный момент для каждого окружения

Чистка FF

Проблема – со временем код обрастает ветками if {} и становится тяжело поддерживаемым

Поэтому нужно регулярно чистить неактуальные флаги

Процесс:

1. Определяем какие фиче-флаги больше не нужны
2. Убираем логику ветвления в коде
3. Убираем FF из FeatureManager в сервисах
4. Деплоим новую версию приложения на все окружения
5. Удаляем FF на портале фичефлагов

Поддержка целостности флага для всех микросервисов

Проблема консистентности при чтении флага из хранилища – если читаем данные всегда из хранилища, то можем получить ситуацию, когда разные сервисы в контексте одного запроса получают разное состояние флага



Поддержка целостности флага для всех микросервисов

Вариант решения

Делаем SDK, который:

- Читает состояние флага
 1. Из хранилища если состояния нет в контексте
 2. Из контекста, если состояние есть в контексте
- Для каждого проверенного флага его значение добавляется в контекст
- Передает контекст между сервисами





Feature Flags сложности

1. Необходимость настройки дополнительной инфраструктуры и усложнение кода
2. С ростом числа флагов и их комбинаций может возникнуть сложность в управлении и отслеживании всех конфигураций
3. Усложняется процесс тестирования из-за большого количества возможных комбинаций включенных флагов



Feature Flags когда оправдано

1. При реализации длинных фич
2. Большой рефакторинг
3. Когда есть зависимость от интеграций с внешними системами
4. Использование для A/B тестирования, отложенных релизов



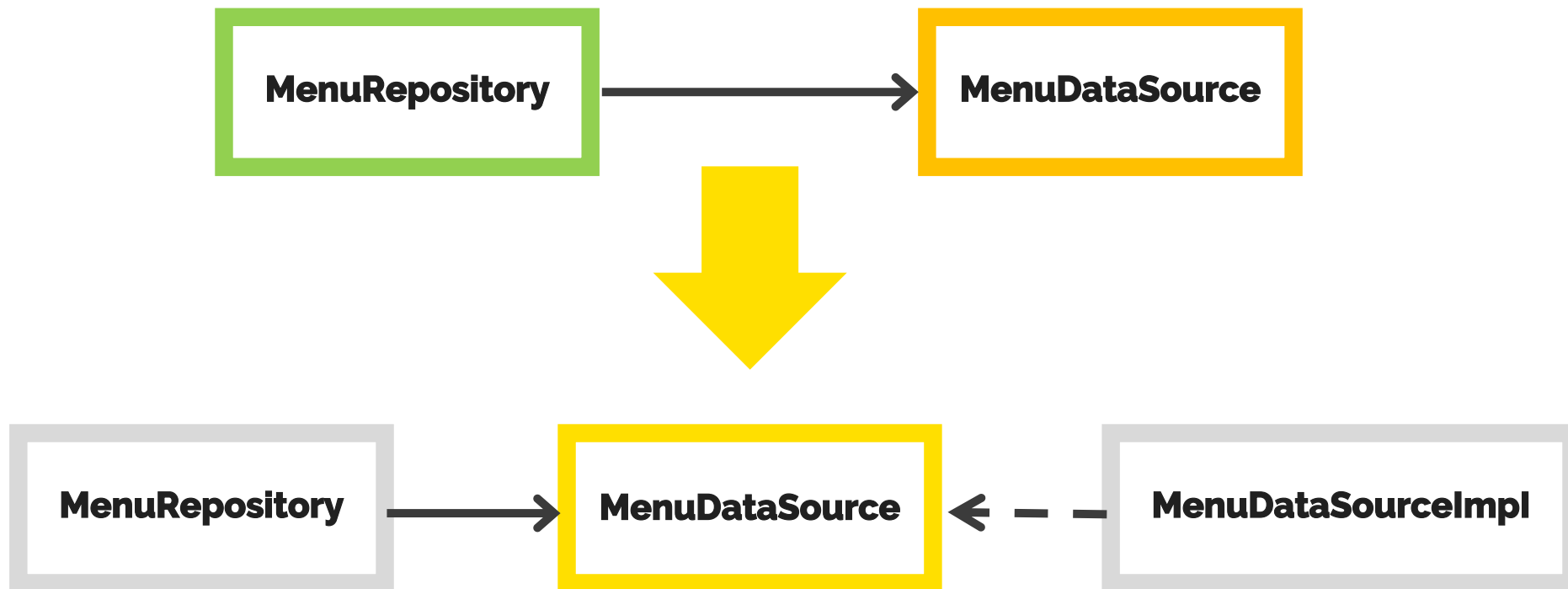
Branch by abstraction

Техника разработки, при которой большие изменения системы делаются постепенно и позволяет релизить приложение до завершения изменений

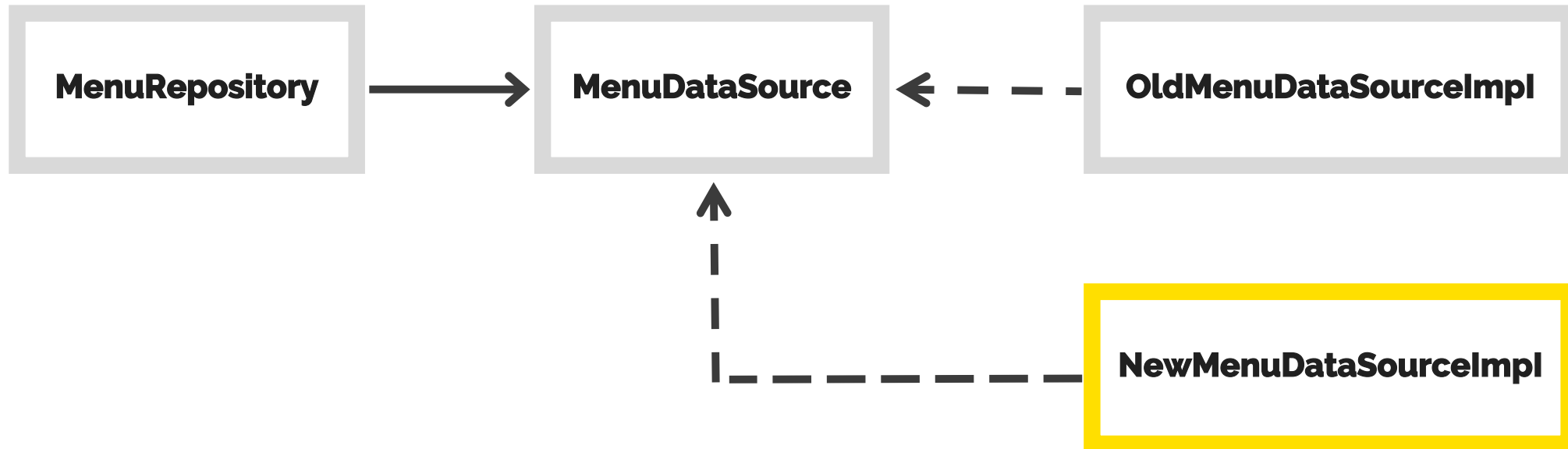
Основные шаги

1. Введение абстракции
2. Создание новой реализации
3. Включить у себя, выключить для остальных
4. Итеративно делаем новую реализацию
5. Удаляем старую реализацию

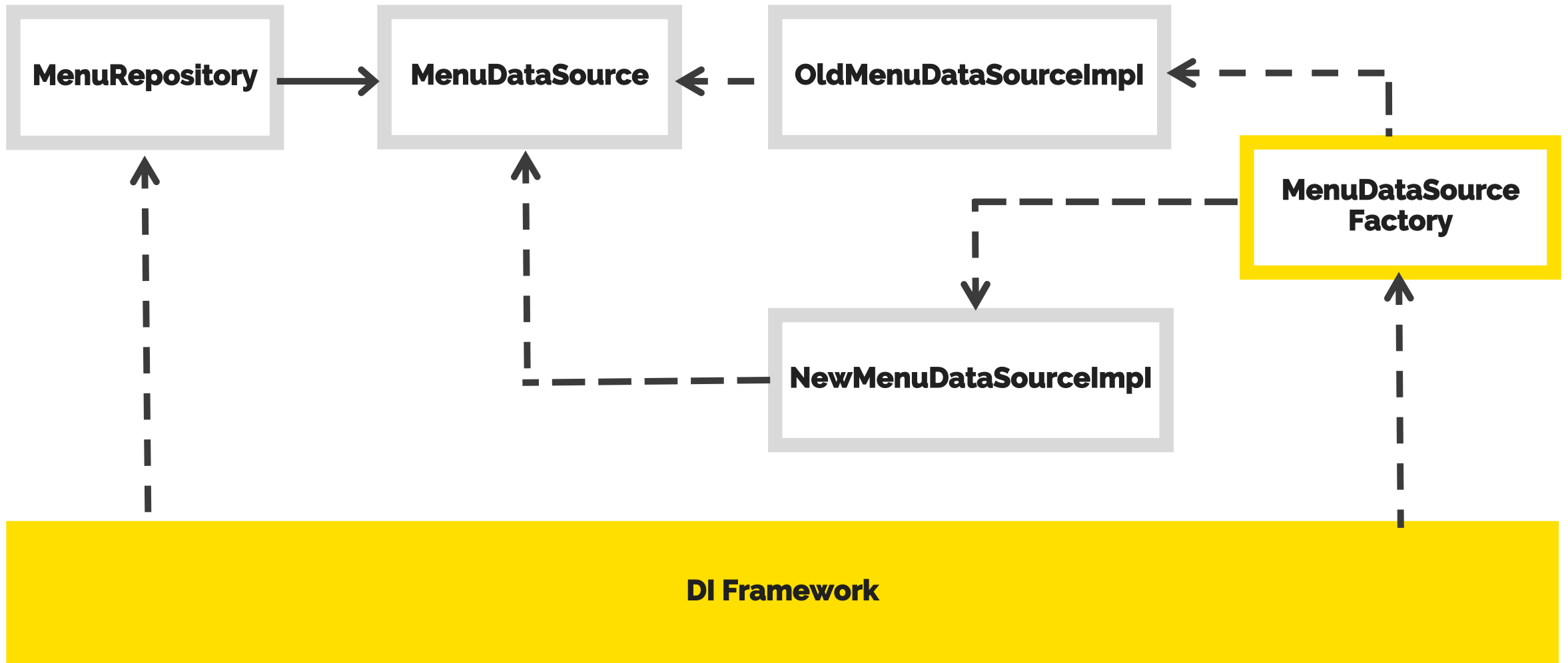
Введение абстракции



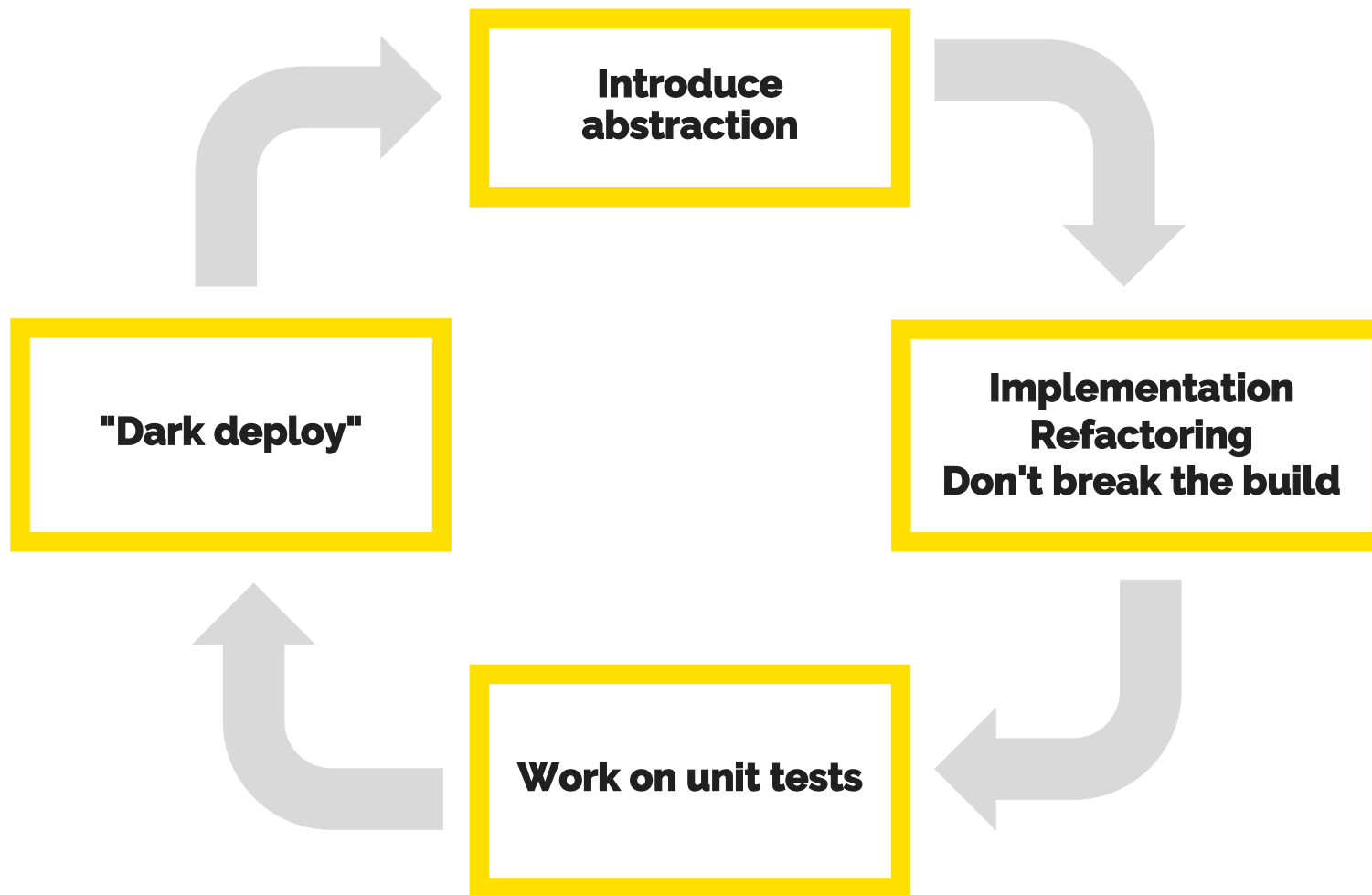
Создание новой реализации



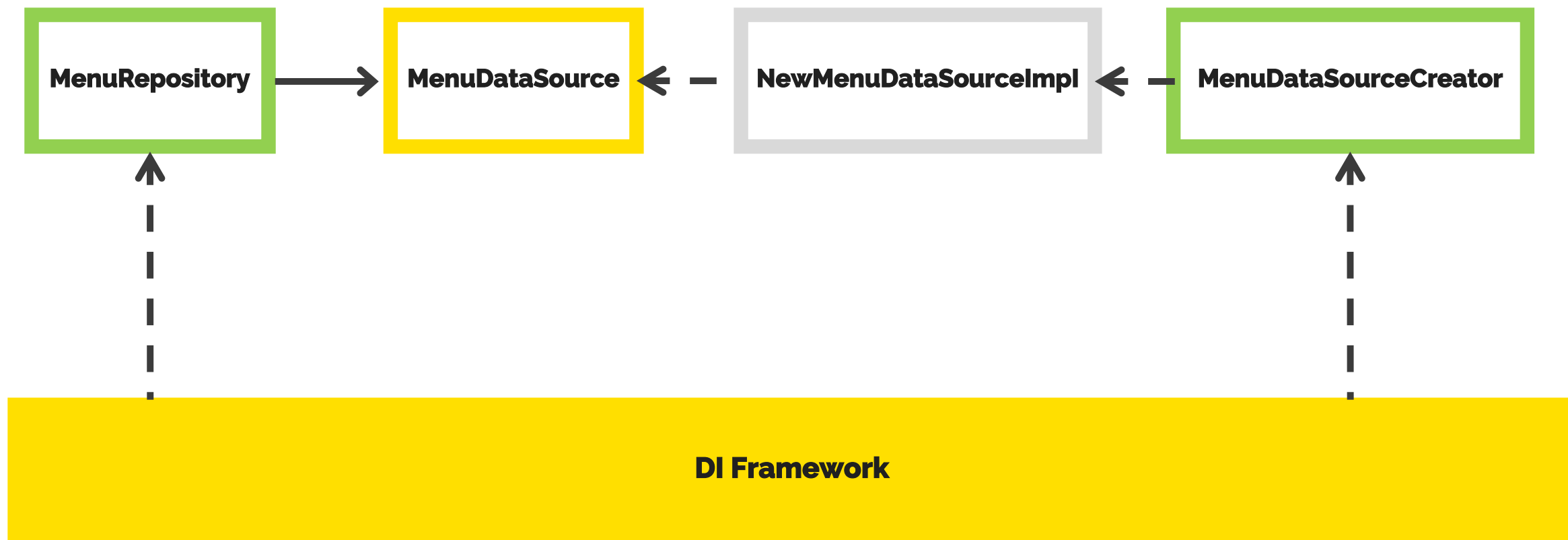
Включить у себя, выключить у остальных




Делаем новую реализацию



Удаляем старую реализацию





Организация репозитория для сложных систем

Ключевые вопросы:

1. Что хранить в репозитории?
2. Работа с секретами
3. Что выбрать моно- или поли-репозиторий?

Что храним в репозитории

Храним то и только то, что необходимо для сборки артефактов. Репозиторий должен быть самодостаточный а сборка проекта воспроизводима

Что храним

1. Исходный код
2. Ресурсы, ассеты
3. Файлы конфигурации
4. Пайплайны для CI/CD
5. Схемы и миграции БД
6. Начальные данные и справочники
7. Данные для генерации документации
8. Автотесты

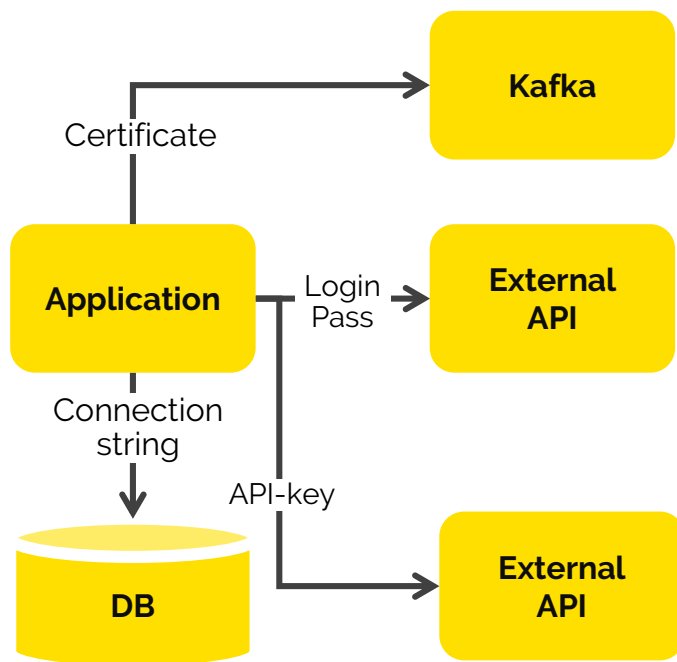
Что **НЕ** храним

1. Артефакты сборки
2. Код библиотек (используем пакетные менеджеры)
3. Секреты

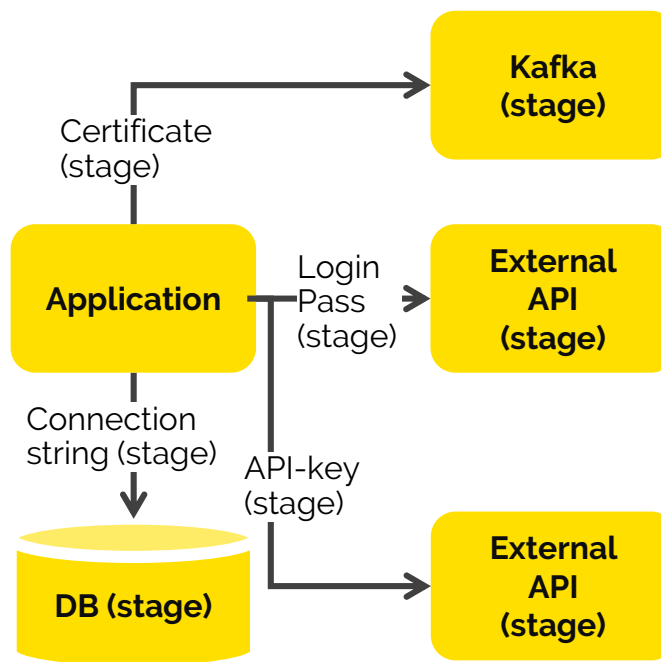
Секреты

Секрет – это конфиденциальные данные, которые дают прямой или косвенный доступ к информационным ресурсам

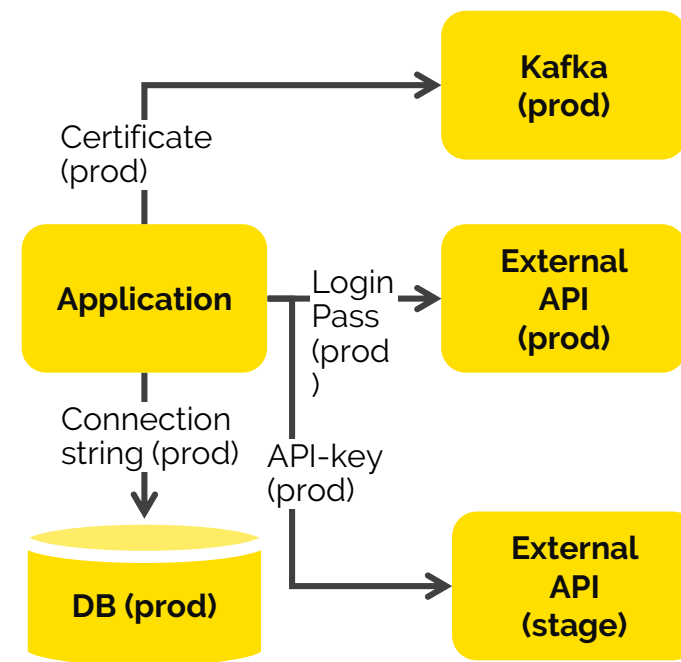
Пример секретов



Stage окружение



Прод окружение





Секреты. Почему плохо хранить?

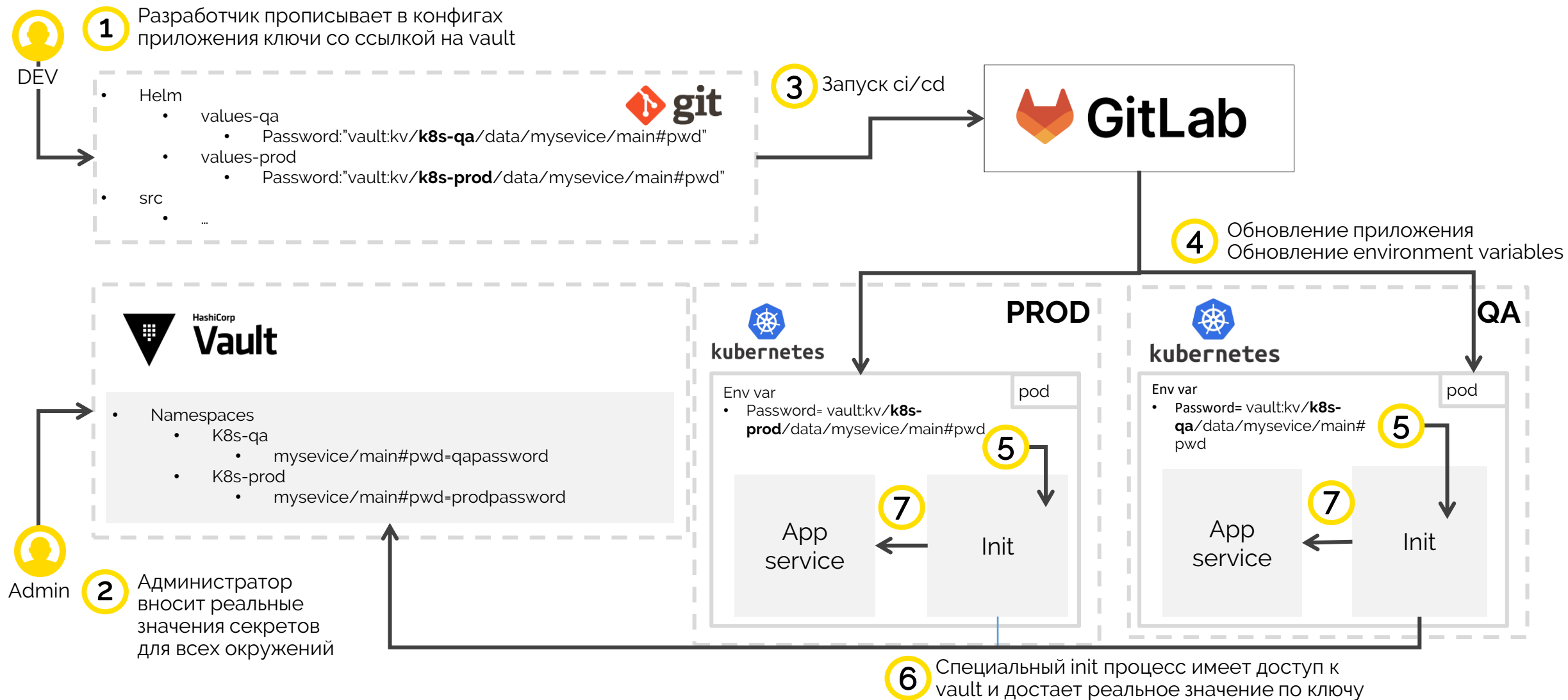
1. Безопасность. Если у команды есть доступ к продуктовым системам – есть риск что-то повредить непреднамеренно или намеренно
2. Доступ к конфиденциальным данным
3. Для изменения пароля придется делать релиз



Управление секретами

1. On-premise менеджеры секретов (Hashicorp vault, Cyberark conjur)
2. Облачные сервисы (Yandex Lockbox, Azure Key Vault)
3. Хранение зашифрованных данных в репозитории + GPG ключ + система управления конфигурацией (например Ansible)
4. Хранение в системе деплоя (gitlab, azure devops, Jenkins)
5. Ручное подкладывание секретов на сервера

Пример организации



Монорепозитории

Стратегия разработки ПО, когда код множества подпроектов хранится в одном и том же репозитории

Преимущества

- 1. Единое управление зависимостями:** Все зависимости и пакеты находятся в одном месте, что упрощает управление версиями и обновлениями.
- 2. Целостность кода:** Все части проекта видны и доступны для всех разработчиков, что способствует лучшему пониманию и сотрудничеству.
- 3. Общий процесс CI/CD:** Легче настраивать и поддерживать единый процесс непрерывной интеграции и доставки для всех компонентов.
- 4. Повышенная консистентность:** Общие правила кодирования и стандарты применяются ко всем частям проекта.
- 5. Упрощенный рефакторинг:** Легче проводить рефакторинг кода, который затрагивает несколько компонентов.

Недостатки

- 1. Масштабируемость:** По мере роста проекта управление одним большим репозиторием становится сложнее.
- 2. Время сборки:** Общий процесс сборки может занимать много времени из-за большого объема кода.
- 3. Разрешения:** Может быть сложно ограничить доступ к определенным частям кода, если это необходимо.
- 4. Конфликты изменений:** С увеличением числа разработчиков возрастает вероятность конфликтов при слиянии изменений.

Поли(мульти)репозитории

Стратегия разработки ПО, у каждого подпроекта свой отдельный репозиторий

Преимущества

1. **Масштабируемость:** Легче управлять отдельными репозиториями, особенно если проект разбит на независимые модули.
2. **Изоляция:** Команды могут работать независимо, не мешая друг другу, и не рискуют случайно сломать чужой код.
3. **Быстрая сборка:** Меньшие репозитории позволяют быстрее собирать и тестировать отдельные части проекта.
4. **Гибкость:** Можно использовать разные инструменты и процессы CI/CD для разных модулей.

Недостатки

1. **Управление зависимостями:** Может быть сложно поддерживать совместимость версий зависимостей между различными модулями.
2. **Трудности интеграции:** Периодическая интеграция модулей может быть сложной и требовать дополнительных усилий.
3. **Повторение кода:** В разных репозиториях может появляться дублирование кода и настроек.
4. **Разрозненность:** Труднее обеспечить единые стандарты и практики по всему проекту.



Выбор стратегии

Монорепозиторий

1. Небольшие и средние команды
2. Небольшой размер проекта
3. Если важна быстрая и частая интеграция
4. Если важен единый контекст команды

Мультирепозиторий

1. Большие команды или отдельные команды для сервисов
2. Большие проекты со множеством микросервисов
3. Если необходимо развертывать и тестировать сервисы отдельно

CI/CD конвейер

CI (continuous integration) - практика разработки ПО, которая заключается в постоянном слиянии рабочих копий в основную ветвь разработки и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем

CD (continuous delivery) - это продолжение CI, которое позволяет автоматически разворачивать успешно собранный и протестированный код на сервере или другой среде реального применения





CI/CD конвейер

Этапы

- Получение кода из системы контроля версий
- Выполнение сборки.
- Прогон unit-тестов
- Проверки SAST (code-style, статический анализ, проверки безопасности)
- Прогон интеграционных тестов
- *Настройка инфраструктуры, автоматизированной через подход "инфраструктура как код".
- Развертывание компонентов приложения (веб-серверы, API-сервисы, базы данных).
- Выполнение дополнительных действий, таких как перезапуск сервисов или вызов сервисов, необходимых для работоспособности новых изменений.
- Выполнение e2e тестов и откат изменений окружения в случае провала тестов.
- Логирование и отправка оповещений о состоянии поставки



CI/CD конвейер

Что нужно для настройки

- Система для автоматизации сборки (gitlab ci/cd, Jenkins, Azure DevOps, TeamCity)
- Серверы (или виртуальные машины) для агентов или раннеров, которые будут выполнять сборку
- Хранилище артефактов сборки (например Container Registry для докер образов)
- Настройка пайплайнов для сборки и развертывания (например в виде yaml скриптов)
- Настройка SAST инструментов (например SonarQube, Trivy, DependencyTrack)
- Тестовые среды для ручного и автоматизированного тестирования



CI/CD конвейер

Преимущества

- проблемы интеграции выявляются и исправляются быстро, что оказывается дешевле;
- немедленный прогон модульных тестов для свежих изменений;
- постоянное наличие текущей стабильной версии вместе с продуктами сборки — для тестирования, демонстрации, и т. п.
- немедленный эффект от неполного или неработающего кода приучает разработчиков к работе в итеративном режиме с более коротким циклом.

Сложности

- Компетенции DevOps в команде
- Значительные затраты на настройку поддержку работы непрерывной интеграции
- Необходимость в дополнительных вычислительных ресурсах под нужды непрерывной интеграции

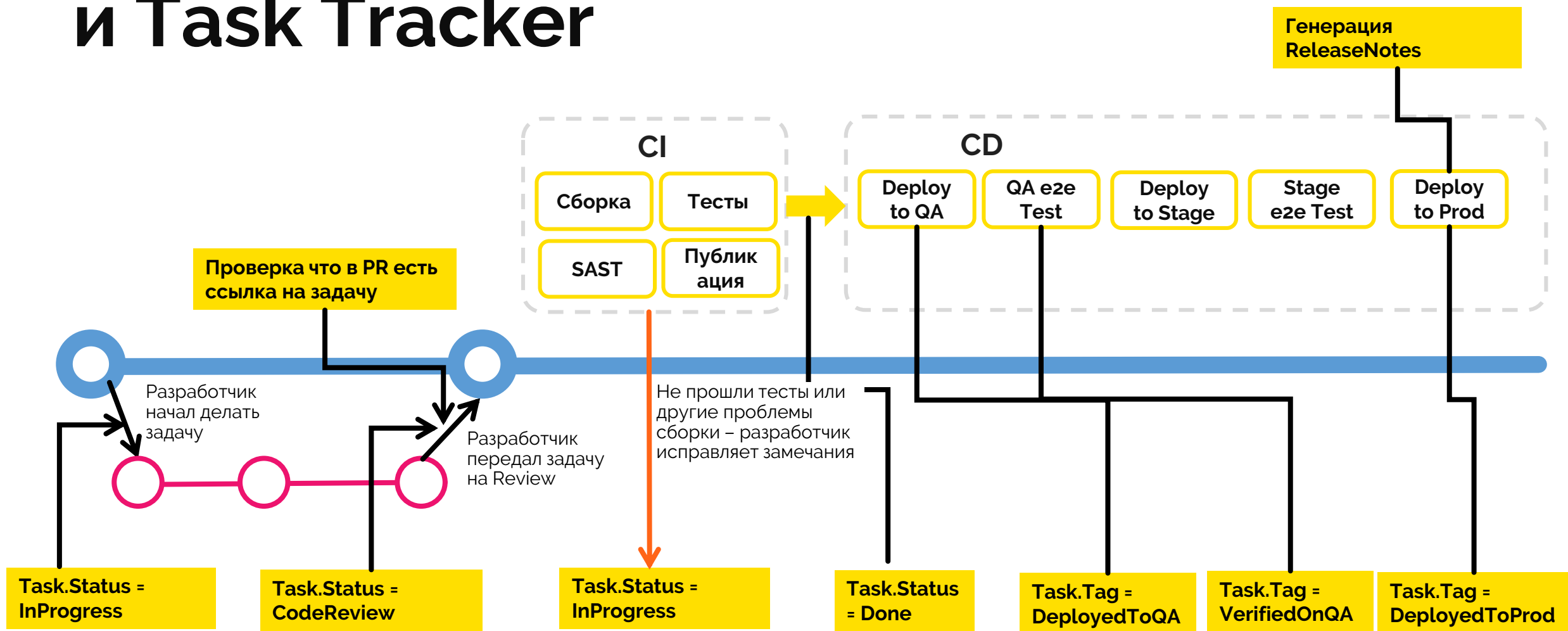
Автоматизация на стыке VCS и Task Tracker

Цель – чтобы текущее состояние задач в Task Tracker отражало актуальное состояние кода и окружений

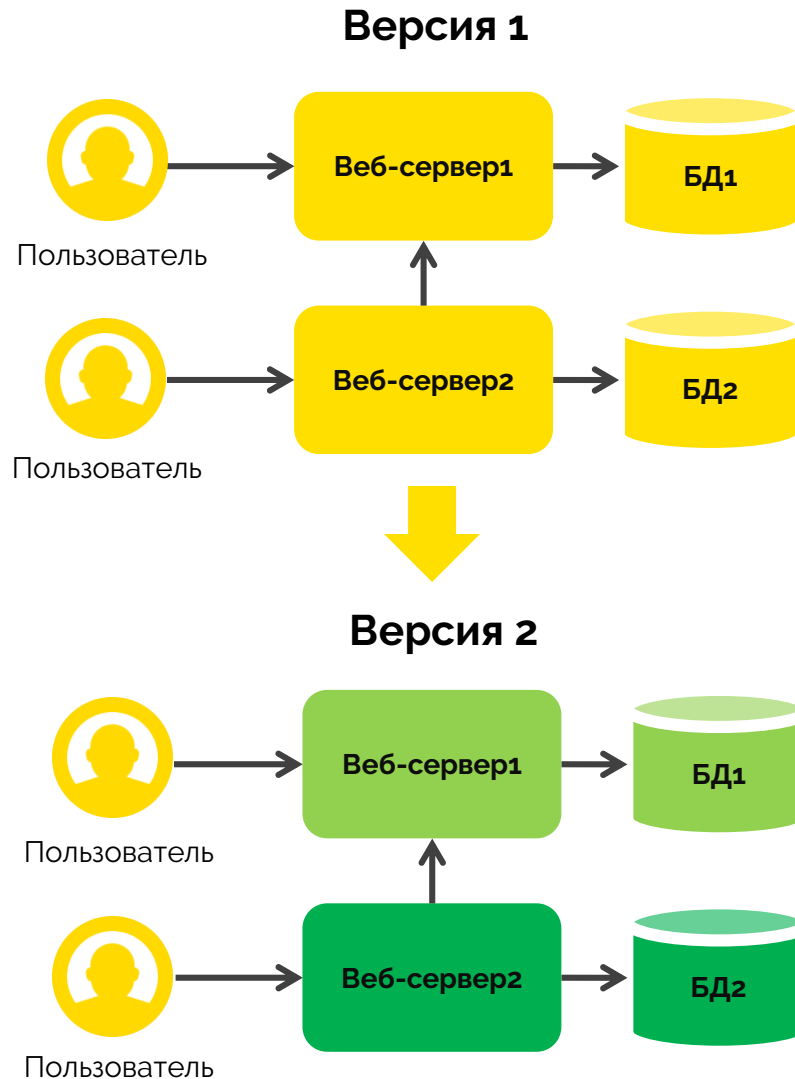
Примеры автоматизации

- Автоматическая смена статусов задач при коммитах и создании Pull Request
- Проставление признака окружения на задаче
- Проверка Pull Request на наличие ссылки на задачу
- Автоматическая сборка Release Notes

Автоматизация на стыке VCS и Task Tracker



Проблемы классического deployment

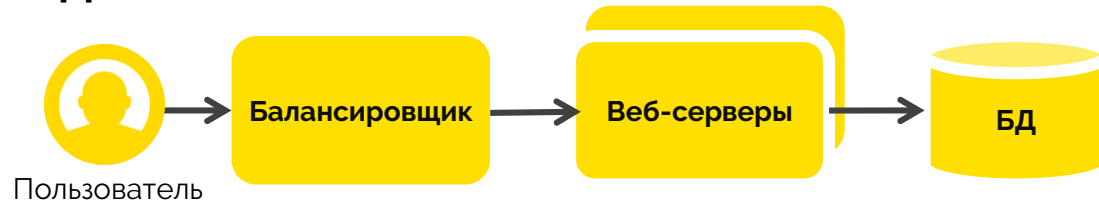


Проблемы подхода

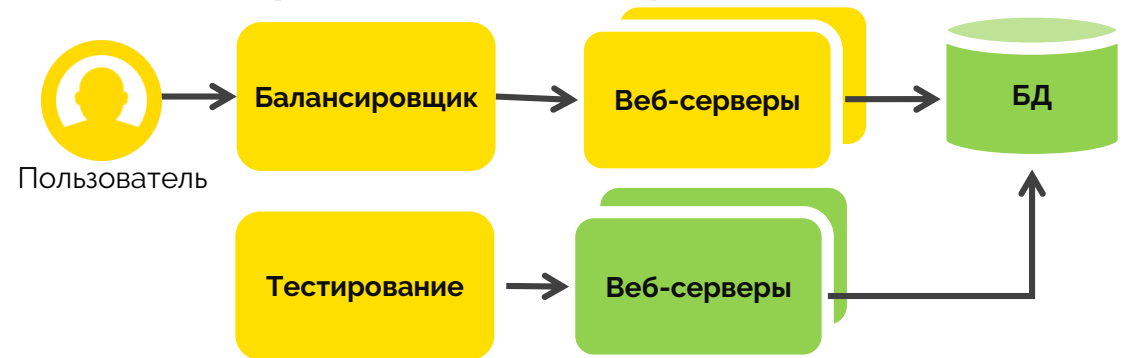
1. Остановка сервера для обновления с версии 1 на версию 2
2. Обновление версии сервера с зависимостями
3. Отказ от новой версии сервера с зависимостями приводит к цепной реакции
4. Для отката нам нужна резервная копия БД
5. При откате повторная остановка сервера
6. При откате теряются данные

Blue green deployment

До обновления

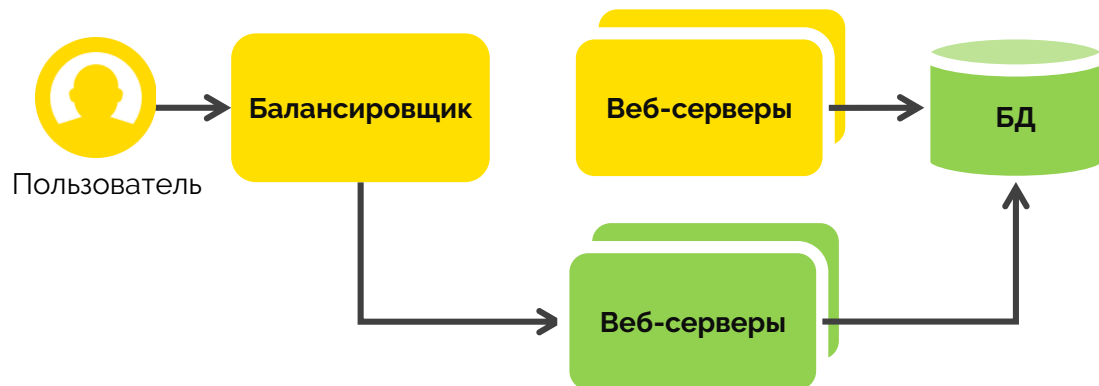


Шаг 1 тестирование новой версии

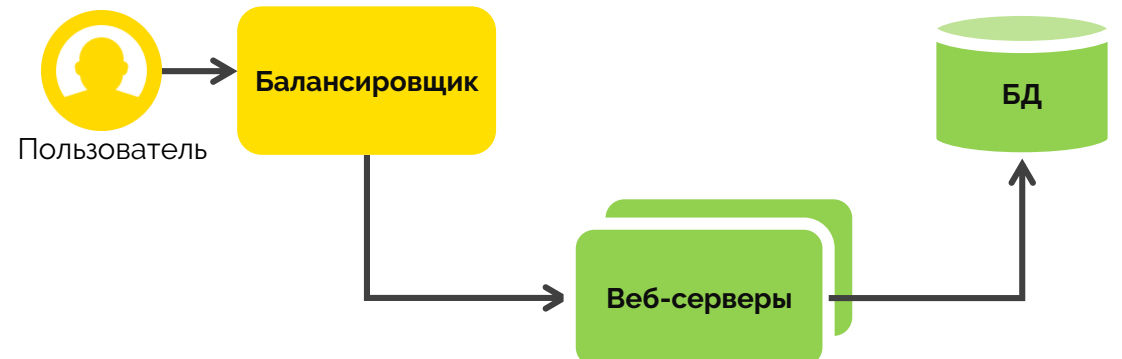


*БД обновляется подходом с обратной совместимостью

Шаг 2 переключение на новую версию

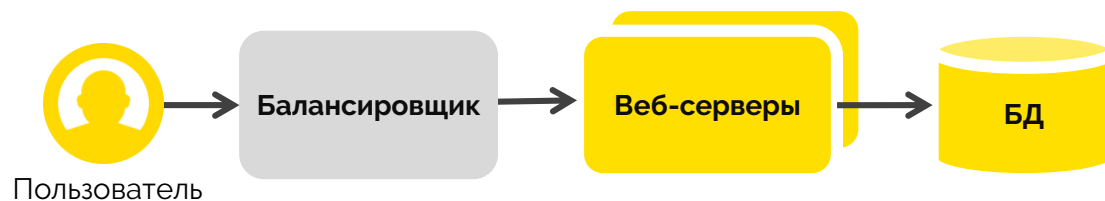


Шаг 3 выключение старой версии

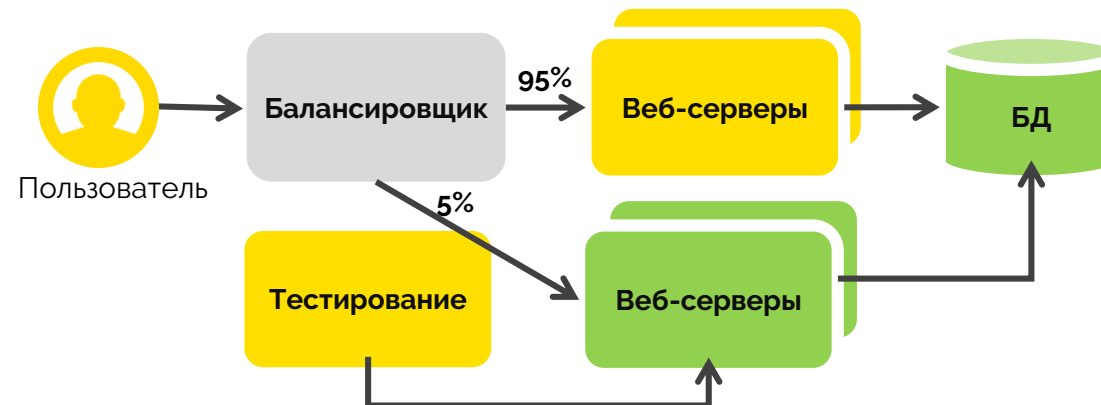


Канареечный релиз

До обновления

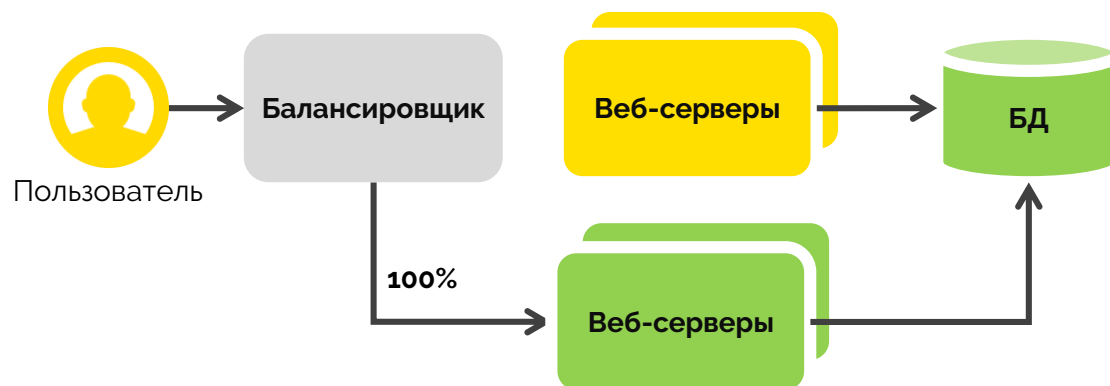


Шаг 1 тестирование новой версии

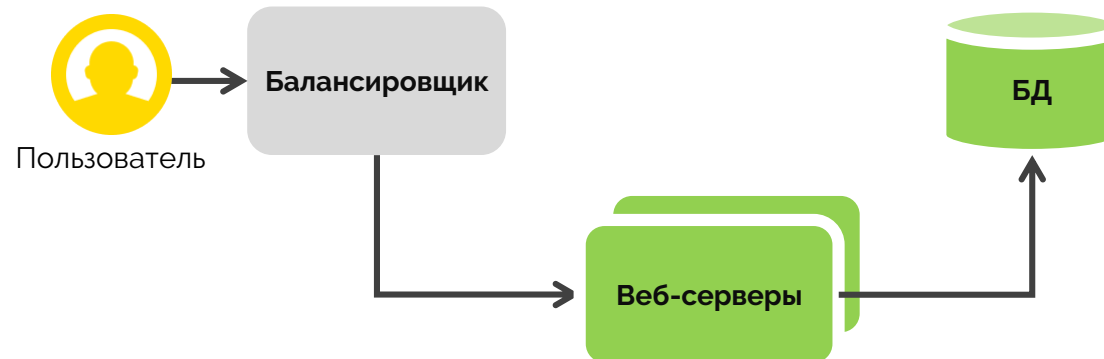


*БД обновляется подходом с обратной совместимостью

Шаг 2 переключение на новую версию



Шаг 3 выключение старой версии





Выбор подхода к CI/CD

Простой процесс

- Проще в реализации, требует меньше компетенций Devops
- Требуется меньше инфраструктуры
- Не подходит если нужна отказоустойчивость 24/7
- Не подходит для высоконагруженных приложений

Blue green / Canary

- Более сложная схема, требуется больше опыта команды
- Нужна дополнительная инфраструктура на инструменты развертывания и на избыточность
- Больше трудозатрат на реализацию и поддержку
- Подходит для критичных сервисов с требованиями к отказоустойчивости

Выбор стратегии: модель ветвления

Критерий	GitFlow	TBD
Типы продуктов и их фазы	Продукт с нуля	Работающий продукт, активно развивающийся
Релизные циклы	Длинные, с предварительными релизами	Короткие, релизы могут быть частыми и быстрыми
Сложность фичей	Поддерживает сложные фичи, требующие длительной разработки	Лучше подходит для простых или средней сложности фич
Опытность команды	Подходит для команд с любым уровнем опыта, но требует хорошей документации процесса внутри команды	Требует экспертизы DevOps, дисциплины CI/CD и поддержки процессов непрерывной интеграции
Стоимость	Относительно дешево	Дорого

N*

**Спасибо
за внимание**

ФИТ Лекция N°4. 25'