НАЗВАНИЕ

Дата выполнения: дата

Выполнил: ФИО

Оглавление

[Введение 2](#__RefHeading___Toc95_1065274009)

[1 Концепция Shift Left 3](#__RefHeading___Toc97_1065274009)

[2 Структура DevSecOps-пайплайна 5](#__RefHeading___Toc99_1065274009)

[2.1 Pre-commit проверки 5](#__RefHeading___Toc101_1065274009)

[2.2 Pre-build проверки 6](#__RefHeading___Toc103_1065274009)

[2.3 Test-time проверка DAST 10](#__RefHeading___Toc105_1065274009)

[2.4 Post-deploy проверки 12](#__RefHeading___Toc329_367963048)

# 

# Введение

DevSecOps — актуальное направление, благодаря которому организации могут быстро создавать и доставлять на рынок безопасные приложения. Этот метод помогает соблюдать регулирующие требования и обеспечивать безопасность на всех этапах цикла разработки и доставки программного обеспечения.

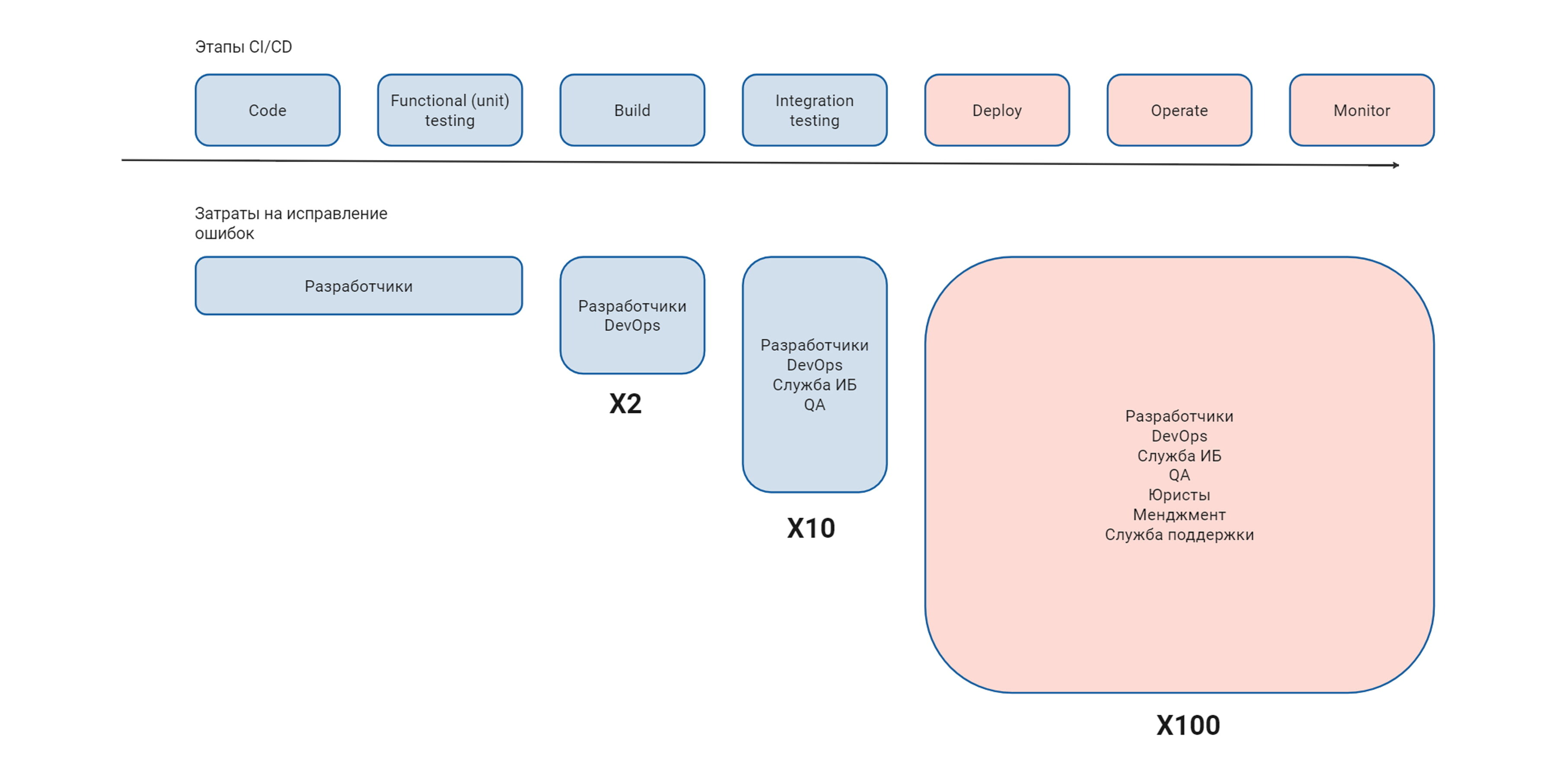
# 

# Концепция Shift Left

Главная цель методологии DevSecOps — внедрение автоматизированных проверок безопасности в DevOps-пайплайн, то есть в сам процесс разработки ПО.

До недавнего времени специалисты по информационной безопасности проводили проверки уже после завершения процесса разработки. Такой подход неэффективен — если в процессе тестирования безопасности обнаруживаются ошибки, приходится начинать весь цикл разработки заново. Это долго и дорого.

Посмотрите схему ниже. На ней видно, что стоимость исправления ошибок растёт с каждым этапом.



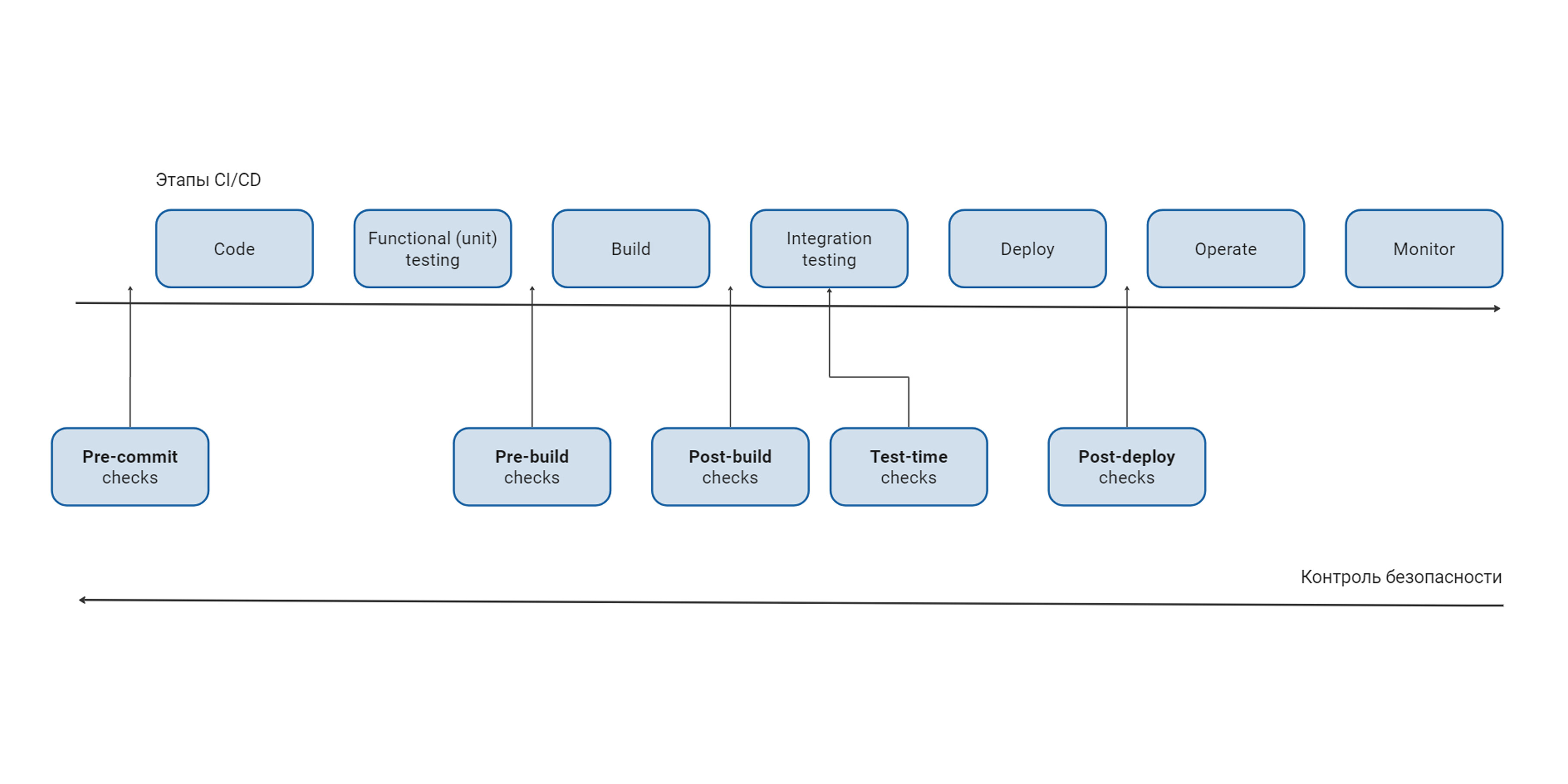
Исправить проблемы с безопасностью, которые обнаружились в процессе разработки и функционального тестирования, почти ничего не стоит. Все нужные изменения можно сразу внести в исходный код и отправить на повторную проверку.

Исправить проблемы на этапе сборки артефактов дороже минимум в два раза. Придётся вносить изменения в процесс сборки, например, редактировать Dockerfile, а значит, потребуется помощь DevOps-инженеров.

Если ошибка обнаружилась во время интеграционного тестирования, её исправление будет в десятки раз дороже. Ведь в этом случае придётся перезапускать цикл разработки заново и привлекать разработчиков, DevOps-инженеров и тестировщиков.

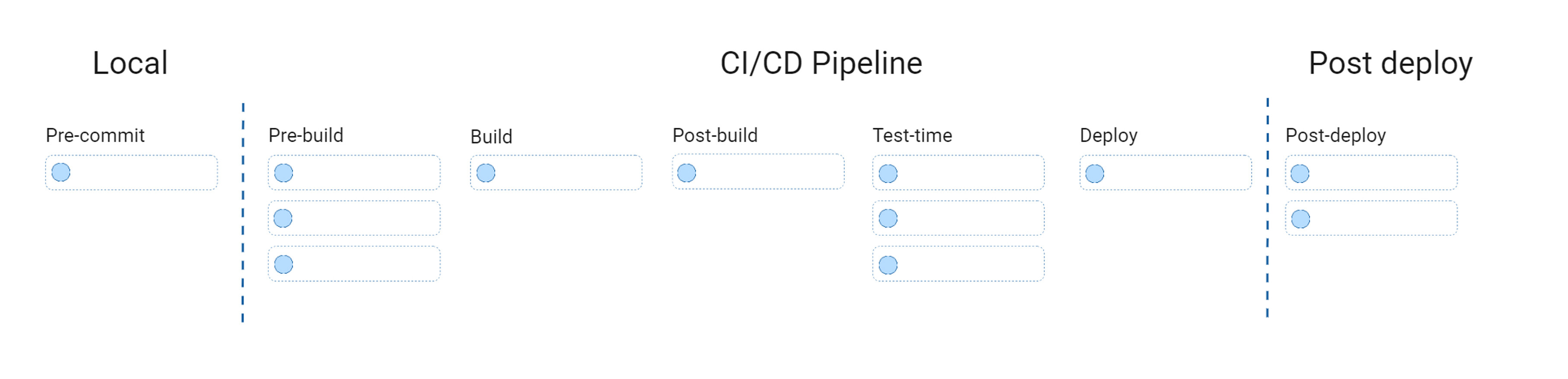
Проблемы с безопасностью, обнаруженные на этапе развертывания, могут привести к утечкам данных пользователей. Компания может получить миллионные штрафы и испортить репутацию, а значит, цена ошибки возрастает в сотни раз.

Отсюда главный вывод — проверки безопасности нужно выполнять как можно раньше. Если представить это визуально, перенести их в левую сторону пайплайна.



# Структура DevSecOps-пайплайна

DevSecOps-пайплайн — это автоматизированная цепочка процессов и инструментов, которые позволяют создавать, тестировать и доставлять приложения в производственную среду, а также обеспечивать их безопасность на каждом этапе.



На рисунке выше изображена структура DevSecOps-пайплайна со всеми основными фазами проверки безопасности. Несколько слов о каждой из них:

Pre-commit — проверка безопасности исходного кода до того, как разработчик закоммитит код в систему контроля версий. Проверка позволяет обнаружить незашифрованную конфиденциальную информацию: пароли или API-ключи.

Pre-build — проверка безопасности исходного кода после того, как он попал в систему контроля версий. Инструменты выполняют статический анализ исходного кода и его зависимостей, чтобы обнаружить распространенные уязвимости, например, многие из списка OWASP Top Ten.

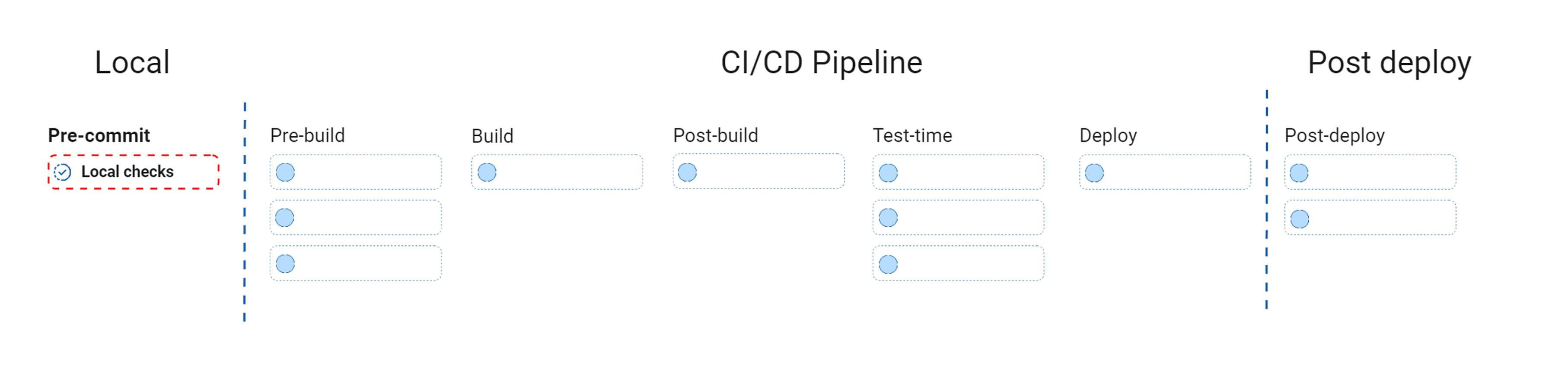
Post-build — проверка безопасности артефакта, собранного из исходного кода, например, Docker-файла, RPM-пакета или JAR-архива. Инструменты анализируют окружение и зависимости приложения, отыскивая устаревшие версии пакетов и библиотек, уже имеющих исправления в более новых версиях.

Test-time — проверка безопасности приложения, которое запущено из собранного артефакта. Инструменты в этой фазе пытаются «сломать» приложение, имитируя действия злоумышленников.

Post-deploy — проверка безопасности приложения после развёртывания в production-среде. Инструменты в режиме реального времени мониторят открытые реестры известных уязвимостей и отправляют уведомление, обнаружив угрозу для приложения.

## Pre-commit проверки

Pre-commit проверки позволяют анализировать исходный код на машине разработчика перед коммитом изменений в локальную копию репозитория. Если какая-то из проверок возвращает ошибку, коммит не выполняется. В этом случае разработчик должен исправить ошибку и заново сделать коммит. Такая проверка позволяет избежать ситуации, когда в репозиторий попадет непроверенный код, например, с незашифрованными секретами.



Чтобы выполнять pre-commit проверки исходного кода, необходимо установить инструменты на локальные машины разработчиков. Такой подход имеет не только плюсы, но и минусы. Например, различия в окружении: разные версии самих инструментов, разные операционные системы и даже архитектуры процессора. Если разработчики используют разные версии инструментов pre-commit, результаты проверки будут различаться, это может затруднить совместную работу. Учитывайте это, внедряя pre-commit проверки в рамках команды или всей компании.

Инструменты Pre-commit

Существует множество опенсорсных инструментов, с помощью которых можно настроить pre-commit проверки:

Gitleaks

git-secrets

Trivy Secret Scanning

Scan

Whispers

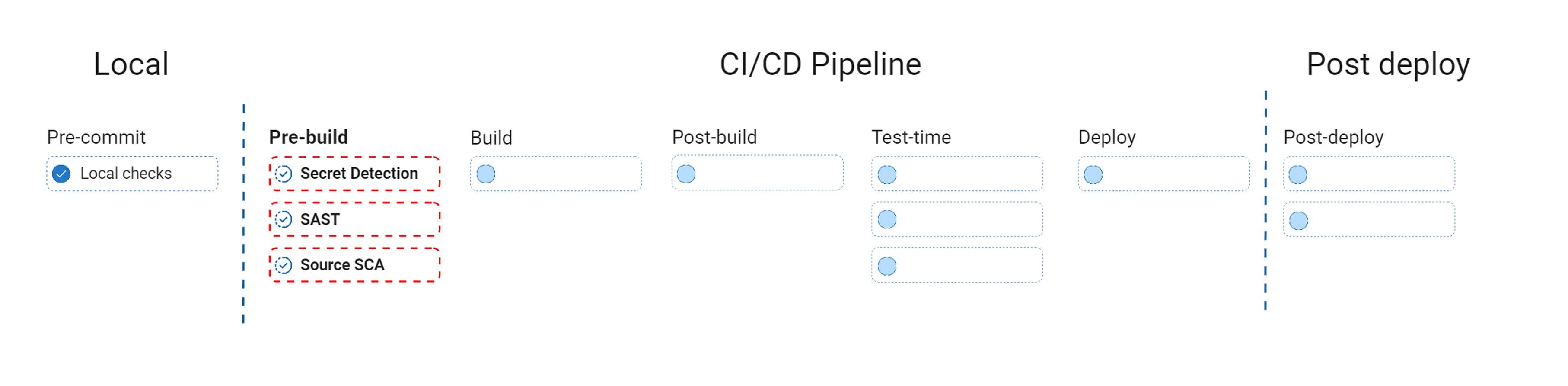
git-all-secrets

detect-secrets

gittyleaks

## Pre-build проверки

На фазе pre-build выполняются проверки методом «белого ящика» (White Box Testing). С помощью таких проверок, как и на предыдущей фазе, анализируется исходный код. В этом случае приложение — «белый ящик», ведь мы знаем и понимаем его архитектуру и внутреннее устройство.



Существует несколько типов pre-build проверок:

Secret Detection;

Static Application Security Testing (SAST);

Software Composition Analysis (SCA).

Pre-build проверка Secret Detection

Secret Detection — это автоматизированная проверка, с помощью которой можно обнаружить незашифрованную конфиденциальную информацию: незашифрованные секреты в исходном коде, попавшем в систему контроля версий.

Pre-build проверки выполняются после того, как исходный код попал в репозиторий системы контроля версий. Поэтому все незашифрованные конфиденциальные данные в репозитории потенциально могут стать доступны злоумышленникам, например в случае утечки содержимого репозитория.

Кроме этого, процесс внедрения secret-detection проверок может происходить не с нуля, а в развивающемся проекте. В этом случае незашифрованные секреты могут обнаружиться в старых коммитах и в различных ветках репозитория.

Чтобы решить эти проблемы, нужно сделать много разных действий. Например, очистить репозитории от незашифрованных секретов, внедрить различные инструменты управления секретами, такие как Hashicorp Vault или Yandex Lockbox.

Инструменты для Secret Detection

Secret Detection можно настроить с помощью Gitleaks, git-secrets или detect-secrets. Но лучше всего использовать сервисы, которые предоставляют известные CI/CD-платформы, например GitLab Secret Detection.

Pre-build проверка SAST

Static Application Security Testing (SAST) — проверка, при которой анализаторы не просто проверяют синтаксическую корректность, но и измеряют качество кода с помощью специальных метрик. Основная задача SAST-сканеров — тестирование на безопасность. В частности SAST-анализаторы проверяют исходный код на наличие распространенных уязвимостей, например, некоторых из списка OWASP Top Ten. Важно сказать, что SAST-сканеры находят не только саму уязвимость, но и фрагмент кода, из-за которого она появилась.

SAST-анализ также называют проверкой методом «белого ящика» (White Box Testing), так как анализатор имеет доступ к внутренней структуре приложения. Важно отметить, что анализаторы проверяют исходный код, не запуская его, поэтому могут генерировать ложные срабатывания и не обнаружить некоторые типы уязвимостей. По этой причине не стоит ограничиваться только SAST-анализом. Лучше подойти к вопросу комплексно и использовать различные типы анализа: SCA, DAST, IAST, OAST.

Инструменты SAST

Бесплатное решение — GitLab SAST

Проприетарные решения:

SonarQube

Checkmarx SAST

Solar appScreener

Российские решения:

PT Application Inspector

PVS-Studio

Pre-build проверка Source SCA

Software Composition Analysis (SCA) — методология, которая обеспечивает безопасность приложений, анализируя их компоненты с открытым исходным кодом.

SCA-сканеры отыскивают устаревшие зависимости, содержащие известные уязвимости и эксплойты. А ещё некоторые SCA-инструменты могут определить лицензионную совместимость компонентов и риски нарушения лицензионных прав.

Source SCA анализирует исходный код, а точнее – зависимости приложения, определенные в исходном коде. Этот анализ часто встречается под именем Dependency Scanning.

SCA позволяет обнаружить проблему, при которой уязвимость в одном компоненте может привести к проблемам с безопасностью во всех приложениях, которые его используют.

Хороший пример — Log4Shell. Эту уязвимость обнаружили в конце 2021 года в Log4j, популярном Java-фреймворке для логирования. Она стала одной из самых страшных уязвимостей, ведь позволяла злоумышленникам выполнять произвольный Java-код на сотнях миллионов устройств.

Инструменты SCA

Опенсорсное решение — Trivy

В составе GitLab (доступно только в Ultimate-версии) — GitLab Dependency Scanning

Российское решение — Profiscope CodeScoring

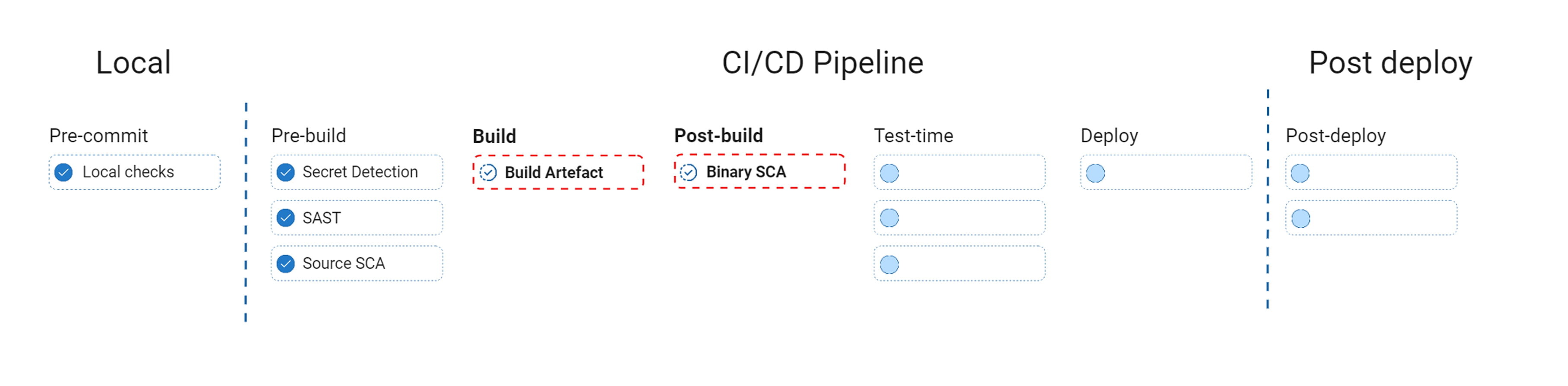
Коммерческое решение с бесплатными тарифными планами:

FOSSA

Mend SCA

Post-build проверки. Binary SCA

Фаза post-build наступает после того, как в CI-пайплайне из исходного кода были собраны артефакты: Docker-образ, RPM-пакет или JAR-архив. С помощью post-build проверок анализируют зависимости в этих бинарных артефактах.

Binary Software Composition Analysis (SCA)

Binary SCA анализ подразумевает проверку бинарных артефактов, таких как Docker-образы, RPM-пакеты и JAR/WAR/EAR-архивы. Его так же часто называют Container Scanning. Запускать его имеет смысл, когда бинарные артефакты собраны, то есть не раньше стадии post-build.

При работе с Docker-образами есть некоторые интересные особенности. Binary SCA-анализаторы проверяют слои Docker-образов и ищут их бинарные слепки в виде хеш-сумм в открытых базах уязвимостей, таких как National Vulnerability Database (NVD) или Snyk Vulnerability DB. Некоторые из анализаторов могут не только отыскать уязвимые компоненты, но и предложить способ исправления, например, указав минимально необходимую версию компонента с уже устранённой уязвимостью.

Примеры популярных Binary SCA анализаторов:

Бесплатное решение — GitLab Container Scanning

Open Source решения:

Trivy

Clair

Grype

Quay

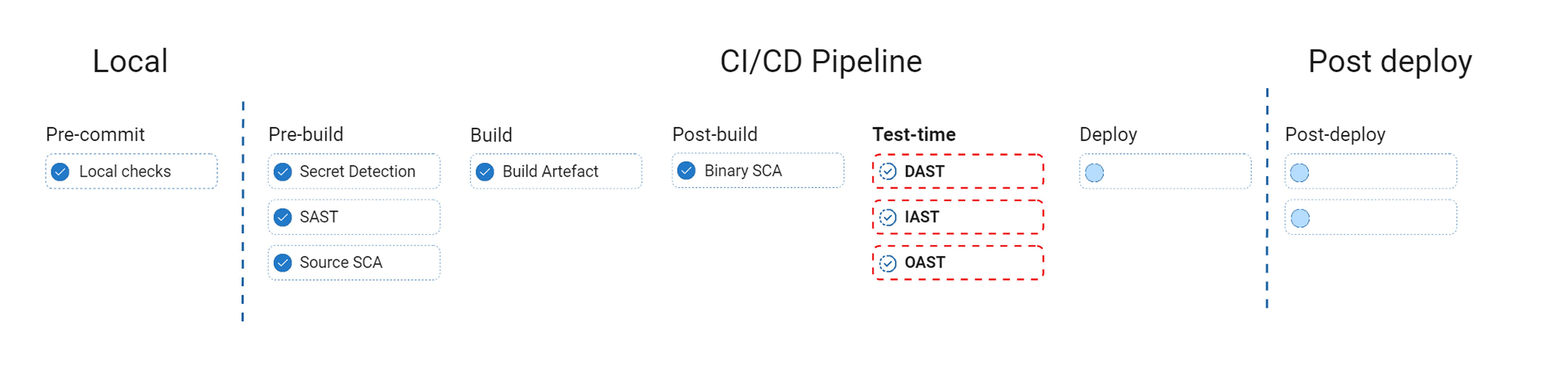
JFrog Xray

Docker Hub

Docker Registry со встроенными анализаторами —Harbour.

Test-time проверки. DAST, IAST и OAST

На этом этапе выполняются динамические проверки методом «серого ящика» (Gray Box Testing) и «чёрного ящика» (Black Box Testing) — когда внутреннее устройство приложения частично или полностью неизвестно. Такие проверки безопасности выполняются с помощью эмуляции действий злоумышленника, который находит уязвимости и пробует «сломать» приложение, воздействуя на них. Далее вкратце рассмотрим каждую из них: DAST, IAST и OAST.



## Test-time проверка DAST

DAST (Dynamic Application Security Testing) — динамическое тестирование безопасности приложений. DAST-сканеры работают автоматически и проверяют приложения, имитируя внешние атаки через различные уязвимости. Получается, что приложение — чёрный ящик для анализатора, о нём ничего не известно.

Для DAST-проверок необходимо иметь доступный для сканера запущенный экземпляр приложения. Причём, чем ближе параметры тестового экземпляра приложения к production-инсталляции, тем меньше будет ложных срабатываний. Например, если вы развернули тестовый экземпляр приложения, доступный только по протоколу http, а в production приложение доступно только по протоколу https, DAST-сканер выдаст ряд ошибок, связанных с отсутствием шифрования трафика между клиентом (анализатором) и сервером (экземпляром приложения).

Примеры DAST-инструментов:

GitLab DAST — доступно только в Ultimate-версии

OWASP Zed Attack Proxy (ZAP) — опенсорсное решение, которое в том числе используется в GitLab DAST

Acunetix

Fortify WebInspect

HCL Security AppScan

Synopsys Managed DAST

Tenable. io (Web App Scanning)

Veracode Dynamic Analysis

Test-time проверка IAST

IAST (Interactive Application Security Testing) — интерактивное тестирование безопасности приложений методом «серого ящика» (Gray Box Testing), соединяющее в себе достоинства и сильные стороны White Box Testing SAST и Black Box Testing DAST.

Чтобы собрать все плюсы и исключить минусы методов SAST и DAST, придумали IAST — гибридный механизм, объединивший эти методы. Он повышает точность динамического тестирования, потому что даёт больше информации о работе приложения благодаря анализу кода.

Стоит учитывать, что помимо плюсов, IAST имеет ограничения. Самое основное — зависимость от языка, на котором написано тестируемое приложение: IAST-решения работают на уровне приложения и требуют доступа к исполняемому коду, чтобы анализировать его поведение. Это означает, что IAST-решения должны быть способными понимать язык программирования, на котором написано приложение. Необходимо отметить, что для одних языков поддержка конкретного IAST-решения может быть реализована лучше, чем для других.

Также IAST-анализ занимает продолжительное время. Оно зависит от многих факторов, например, размера и сложности приложения, количества внешних зависимостей и производительности используемого IAST-инструмента.

Кроме этого, в процессе анализа не сканируется сам код, а проверяются только те фрагменты, которые непосредственно исполняются. Поэтому IAST-анализ лучше всего совместить с функциональным тестированием, когда можно проверить максимально возможное количество сценариев работы приложения.

Примеры IAST-инструментов

Synopsys Seeker

Veracode Interactive Analysis

Microfocus Fortify WebInspect

Checkmarx CxIAST

Contrast Assess

Hdiv Detection (IAST)

Positive Technologies Application Inspector.

Test-time проверка OAST

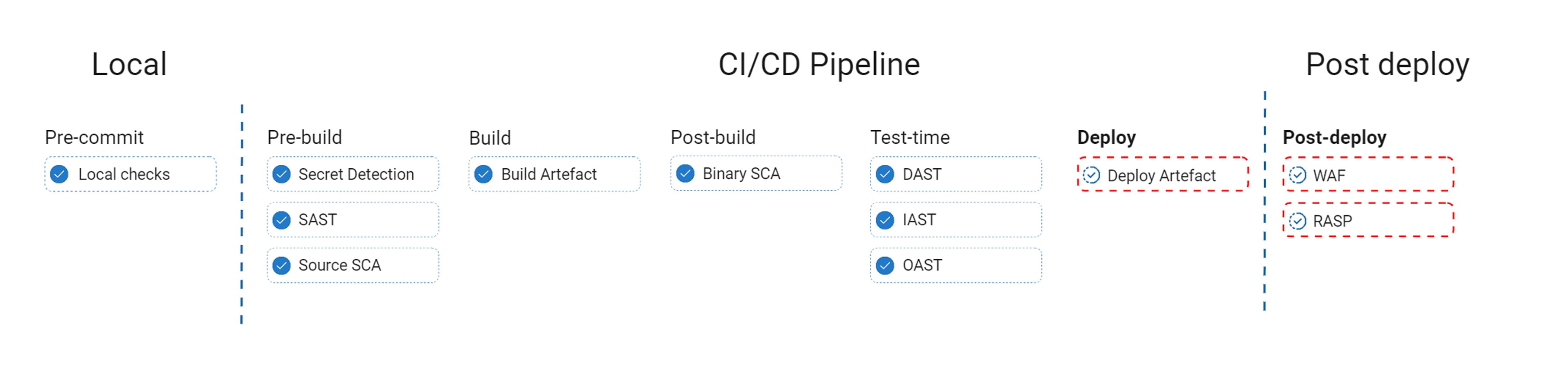
OAST (Out-of-band Application Security Testing) — техника разработана компанией PortSwigger и является расширением DAST, а именнно, находит уязвимости, которые не видит DAST, например, log4shell.

Примеры OAST-инструментов

Burp Suite

OWASP ZAP с OAST-плагином и OAST-сервисами BOAST, TukTuk и interactsh

## Post-deploy проверки



Это важный этап в обеспечении безопасности и работоспособности приложений. В отличие от pre-commit проверок, которые выполняются на этапе разработки, post-deploy-проверки позволяют выявить возможные проблемы уже на этапе эксплуатации приложения, когда оно находится в реальной среде.

Мониторинг безопасности артефактов

Чтобы вовремя обнаружить новые уязвимости, проводить регулярные проверки артефактов необходимо в том числе после деплоя приложения. Это можно делать с помощью специальных хранилищ или инструментов, например, Snyk Container, Trivy, GitLab Container Scanning, или разработки собственного интеграционного модуля.

Мониторинг безопасности приложения

Ещё один способ обеспечить безопасность — мониторинг самого приложения. Для этого существует несколько технологий.

Web Application Firewall (WAF) — межсетевой экран для веб-приложений. Это инструмент для фильтрации трафика. Он работает на прикладном уровне и защищает веб-приложения, анализируя трафик HTTP/HTTPS.

RASP (Runtime Application Self-Protection) — технология, которая в реальном времени обнаруживает и блокирует атаки на приложение. Она добавляет функцию защиты в среду исполнения, что даёт приложению возможность самозащиты (self-protection) в автоматическом режиме.

Главное преимущество технологии RASP перед другими средствами защиты веб-приложений в том, что она понимает, как работает приложение, и выявляет атаки и нелегитимный трафик. При этом RASP может обнаруживать не только типовые атаки, используя метод сравнения по сигнатуре, но и попытки эксплуатации zero-day уязвимостей, реагируя на аномалии.

В отличие от WAF, RASP работает внутри приложения и вместе с ним. WAF можно обмануть. Если злоумышленник изменит код приложения, WAF не сможет это определить, потому что не понимает контекст исполнения. А RASP имеет доступ к диагностической информации о работе приложения, может анализировать её, выявлять аномалии и блокировать атаки.

Технология RASP имеет особенность — фокус на предотвращении атак по умолчанию. Системе не требуется доступ к исходному коду.

RASP-инструменты

Fortify

Sqreen

OpenRASP — опенсорсное решение, разработанное Baidu

Signal Sciences

Jscrambler

Hdiv

Imperva

Визуализация результатов работы DevSecOps-пайплайна

На разных этапах пайплайна мы используем большое количество Application Security Testing/DevSecOps-анализаторов. Каждый из них формирует отчёт в собственном формате, а некоторые генерируют среди найденных уязвимостей так называемые False Positives — ложные срабатывания. Из-за этого анализировать безопасность приложения в целом достаточно сложно.

Чтобы эффективно анализировать уязвимости и управлять процессом их устранения используют специализированные Vulnerability Management инструменты, которые зачастую называют просто Security Dashboards.

Security Dashboards решают проблему, передавая результаты работы DevSecOps-анализаторов в унифицированном и удобном для анализа виде. Так инженеры по безопасности наглядно видят, какие существуют проблемы, какие из них наиболее критичные и какие необходимо устранить в первую очередь.

