УДК 004.01

Брайловский Андрей Валерьевич
Brailovskii Andrei Valerevich
Ассистент
Assistant
МИРЭА - Российский технологический университет
MIREA - Russian University of Technology
Москва, Россия
Moscow, Russia

ОГРАНИЧЕНИЯ И ВЫЗОВЫ ПАРАДИГМЫ «ДОКУМЕНТАЦИЯ КАК КОД»: КРИТИЧЕСКИЙ АНАЛИЗ СИСТЕМ КОНТРОЛЯ ВЕРСИЙ В УПРАВЛЕНИИ ДОКУМЕНТАЦИЕЙ

Аннотация. В статье проводится критический анализ парадигмы «Документация как код» (DaC) с фокусом на ограничениях, которые возникают при применении систем контроля версий (VCS) для управления технической документацией. Детально рассматриваются такие проблемы, как риск рассинхронизации между исходным кодом и документацией, отсутствие встроенных механизмов отслеживания этой связи, инструментальный барьер для нетехнических специалистов, а также непригодность стандартных инструментов VCS для рецензирования текста и управления нетекстовыми файлами. В статье исследуется, каким образом слепое копирование практик разработки ПО без их адаптации может приводить к снижению качества документации и эффективности командной работы. Статья предлагает рассматривать DaC не как готовое решение, а как основу для построения комплексной экосистемы, целенаправленно решающей выявленные проблемы.

Ключевые слова: Документация как код, DaC, ограничения VCS, рассинхронизация документации, системы контроля версий, Git, технический долг, рецензирование документации, управление нетекстовыми файлами,

DevOps.

Abstract. This article provides a critical analysis of the "Docs as Code" (DaC) paradigm, focusing on the limitations that arise when applying version control systems (VCS) to manage technical documentation. It examines in detail such problems as the risk of desynchronization between source code and documentation, the lack of built-in mechanisms to track this connection, the tool-chain barrier for non-technical specialists, and the unsuitability of standard VCS tools for reviewing text and managing non-text files. The article explores how blindly copying software development practices without adaptation can lead to a decrease in documentation quality and team efficiency. It proposes that DaC should be viewed not as a turnkey solution, but as a foundation for building a comprehensive ecosystem that purposefully addresses the identified problems.

Keywords: Docs as code, DaC, VCS limitations, documentation desynchronization, version control systems, Git, technical debt, documentation review, non-text file management, DevOps.

Введение

Признание систем контроля версий (VCS) в качестве фундаментального решения многих проблем традиционного жизненного цикла документации является важным шагом на пути к повышению эффективности и качества. Однако при переходе от теоретических преимуществ к практической реализации команды часто сталкиваются с целым рядом непредвиденных трудностей. Проблема заключается в явлении, которое можно охарактеризовать как «несоответствие инструмента и задачи»: система, идеально спроектированная и отточённая для управления исходным кодом, требует значительной адаптации и доработки для эффективной работы с документацией [1]. Простое внедрение Git в рабочий процесс технических писателей является лишь частью решения. Без правильной настройки, дополнительных инструментов и переосмысления процессов, преимущества нового подхода могут быть нивелированы его сложностью, неудобством и внутренним сопротивлением команды [4]. Данная статья посвящена именно этому практическому аспекту: как распознать и

преодолеть ограничения стандартных VCS и построить на их основе по-настоящему эффективную, удобную и мощную экосистему для управления документацией как кодом.

Проблема рассинхронизации кода и документации

Одно из ключевых «обещаний» подхода DaC — поддержание документации в актуальном состоянии за счёт её сближения с кодом. Однако именно здесь кроется одно из самых глубоких заблуждений. Системы контроля версий, такие как Git, превосходно отслеживают изменения в файлах, но они абсолютно «слепы» к семантическому содержанию этих файлов [2]. Git не имеет никакого представления о том, что функция calculate_interest() в файле billing.py логически связана с разделом «Расчёт процентов» в файле manual.md. Эта связь существует только в головах разработчиков и технических писателей.

Как следствие, в методологии DaC отсутствует какой-либо встроенный механизм, который бы автоматически отслеживал актуальность документации при изменении соответствующего кода. Если разработчик добавляет новый обязательный параметр в функцию, исправляет логику или меняет возвращаемое значение, ничто не заставит его и не напомнит ему внести правки в документацию. Этот процесс остаётся полностью ручным и целиком зависит от дисциплины и коммуникации внутри команды. На практике, в условиях сжатых сроков, разработчики часто забывают обновить документацию или откладывают это «на потом», но «потом» так никогда и не наступает [5].

Это приводит к накоплению очень опасного вида технического долга — «скрытого долга» в документации. В отличие от очевидных пробелов (когда документация просто отсутствует), здесь мы имеем дело с ситуацией, когда документация выглядит полной и актуальной, но на деле содержит устаревшую, вводящую в заблуждение информацию. Пользователь, следующий такой инструкции, может столкнуться с необъяснимыми ошибками, потерять данные или не суметь воспользоваться критически важной функциональностью. Выявление таких расхождений требует проведения дорогостоящего ручного аудита, что сводит на нет многие преимущества от автоматизации. Таким

образом, хотя VCS решает проблему версионирования файлов, она не решает и даже может маскировать более глубокую проблему синхронизации смысла между кодом и текстом.

Инструментальный барьер и когнитивная нагрузка

Второй серьёзный вызов заключается в высоком пороге входа для всех участников процесса, не являющихся разработчиками. Качественная документация создаётся не только техническими писателями; огромный вклад вносят менеджеры по продукту, бизнес-аналитики, профильные эксперты (SME) и редакторы [8]. Для этих специалистов рабочий процесс, построенный на основе Git и сопутствующих инструментов, может оказаться непреодолимым барьером.

Проблема не ограничивается необходимостью изучения десятка команд Git. Она заключается в требовании освоить всю экосистему инструментов разработки: работать в средах вроде VS Code, пользоваться интерфейсом командной строки, разбираться в сложных моделях ветвления (таких как Gitflow), понимать концепцию Pull Request'ов и так далее [2]. Для человека, чья основная задача — формулировать требования, проверять логическую непротиворечивость или вычитывать текст на стилистические ошибки, эти требования являются избыточными и отвлекающими.

В результате возникает парадоксальная ситуация: система, призванная улучшить совместную работу, на деле отсекает от неё ключевых участников [7]. Вместо того чтобы оставить комментарий непосредственно в системе, менеджер по продукту отправляет свои замечания по почте. Редактор выгружает текст, правит его в привычном Word и присылает обратно. Профильный эксперт, вместо того чтобы быстро поправить пару предложений в ветке, описывает суть правок в мессенджере. Весь этот ценный вклад затем должен быть вручную перенесён в Git техническим писателем, который снова становится «бутылочным горлышком». Это не только увеличивает его нагрузку, но и возвращает процесс к тому самому хаосу с множеством источников правок, от которого и пытались уйти [6]. Таким образом, инструментальный барьер не просто создаёт неудобства — он ломает саму идею единого, прозрачного

рабочего потока, что наглядно продемонстрировано на Рисунке 1.

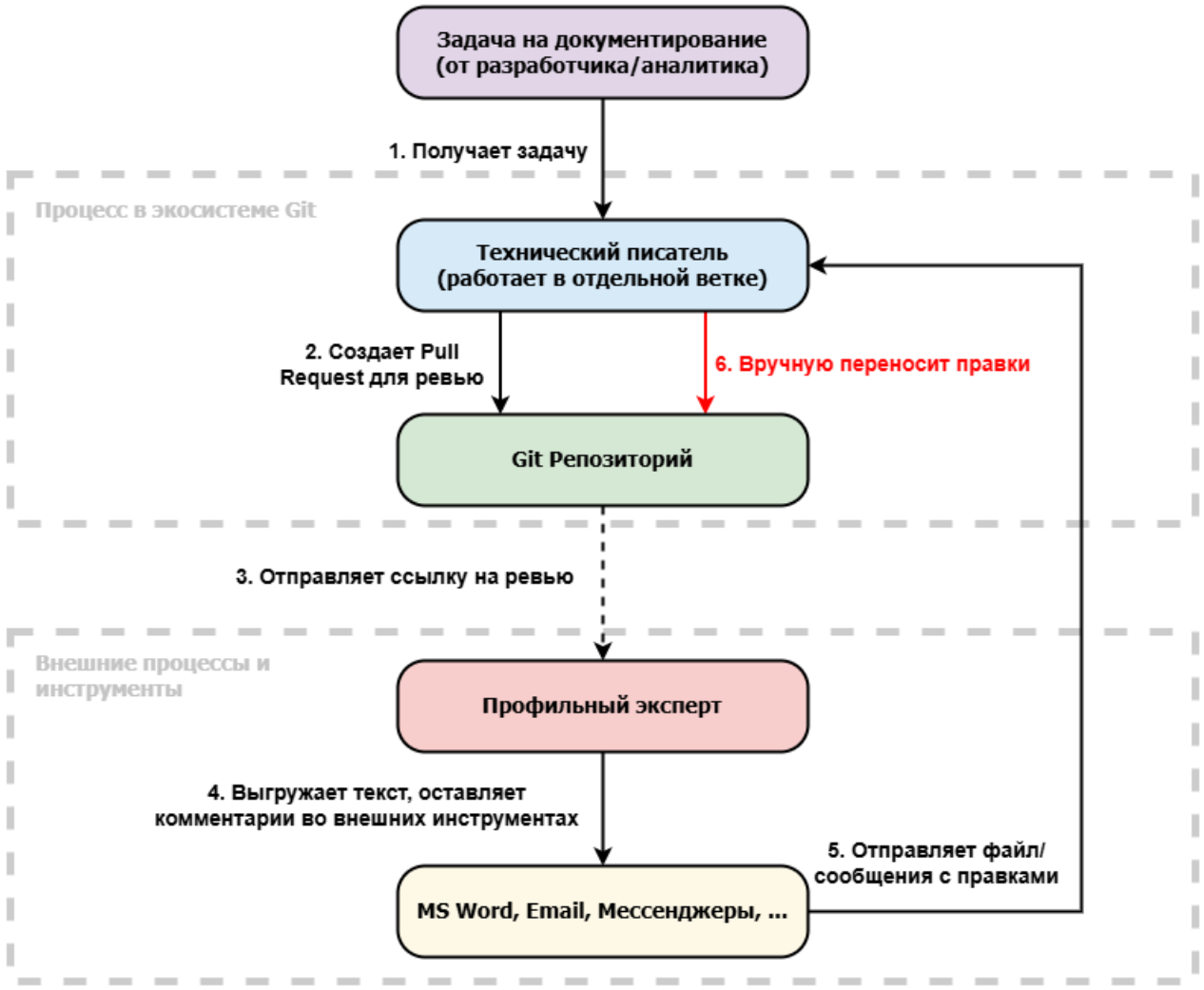

Рисунок 1 – Схема нарушения рабочего процесса из-за инструментального барьера

Непригодность стандартных инструментов для работы с текстом

Помимо высокого порога входа, сами инструменты, предлагаемые VCS, плохо подходят для основной задачи, связанной с документацией, — работы с человеческим языком.

Наиболее ярко это проявляется в механизме сравнения версий (diff). Он был создан для кода и работает построчно [1]. Для кода это логично, но для текста — губительно. Если автор переформулировал длинное предложение, поменяв слова местами, или объединил два коротких предложения в одно, для diff это будет выглядеть как полное удаление старых строк и добавление новых. Разобраться в сути стилистической или структурной правки по такому представлению практически невозможно. Рецензент вынужден тратить время и

когнитивные усилия не на оценку качества текста, а на «дешифровку» вывода инструмента. Это делает процесс рецензирования медленным, утомительным и неэффективным.

Отсутствие привычных инструментов для совместной работы с текстом — ещё одна серьёзная проблема. В таких системах, как Google Docs или Microsoft Word, пользователи могут оставить комментарий к конкретному слову, предложить замену, отследить исправления в режиме рецензирования [4]. В стандартном интерфейсе Pull Request на GitHub или GitLab ничего подобного нет. Комментарии можно оставлять только к целым строкам «сырой» разметки, что крайне неудобно для обсуждения нюансов формулировок.

Поиск в репозитории также представляет собой проблему. Стандартные средства, такие как git grep, осуществляют мощный, но чисто текстовый поиск. Они не способны выполнять семантические запросы, которые необходимы для полноценной базы знаний. Например, невозможно найти все разделы, помеченные как «предназначенные для начинающих», или все предупреждения о безопасности, актуальные для определённой версии продукта. Для реализации такого функционала требуются внешние, более сложные системы индексации и поиска, что дополнительно усложняет экосистему.

Проблема управления нетекстовыми и сложными файлами

Техническая документация редко состоит исключительно из текста. Изображения, диаграммы, схемы, видео и другие мультимедийные файлы играют в ней критически важную роль, а иногда несут основную смысловую нагрузку [8]. Стандартный Git крайне неэффективен при работе с такими бинарными файлами, особенно если они большие. Поскольку Git сохраняет полную копию каждой версии каждого файла, включение даже нескольких десятков скриншотов в репозиторий приводит к его неконтролируемому раздуванию. Это значительно замедляет такие базовые операции, как клонирование репозитория или переключение между ветками, делая работу некомфортной [1].

Для решения этой проблемы было создано расширение Git LFS (Large File

Storage), которое заменяет большие файлы в репозитории небольшими текстовыми указателями. Однако Git LFS не является панацеей, а скорее «костылём», который вносит собственный уровень сложности. Он требует наличия отдельного LFS-сервера, а также установки и правильной настройки клиента у каждого участника команды. Ошибки в конфигурации могут приводить к потере файлов или порче репозитория.

Ещё более сложная ситуация возникает с редактируемыми диаграммами и схемами. Если диаграмма создана во внешнем графическом редакторе (например, Visio, Figma или draw.io) и сохранена в репозитории как изображение (PNG, SVG), то её исходный, редактируемый файл остаётся за пределами системы контроля версий. Чтобы внести малейшее изменение в такую диаграмму — поправить одну надпись или передвинуть один блок — необходимо найти исходный файл, открыть его в соответствующей программе, внести правки, заново экспортировать изображение, заменить старый файл в репозитории и закоммитить новый. Этот процесс не только трудоёмок, но и крайне подвержен ошибкам. Он также полностью исключает возможность совместной работы над одной схемой. Решением этой проблемы является подход «диаграммы как код» (Diagrams as Code) с использованием инструментов вроде PlantUML или Mermaid, которые описывают диаграммы текстовыми командами. Однако это снова возвращает нас к проблеме высокого порога входа, так как требует от технических писателей освоения ещё одного специализированного синтаксиса.

Семантическое версионирование и управления релизами

Системы контроля версий превосходно справляются с версионированием отдельных файлов, но они не предоставляют встроенных инструментов для управления жизненным циклом документации как целостного продукта. В мире ПО принято семантическое версионирование (SemVer), где номер версии (например, 2.1.5) несёт информацию о характере изменений (мажорный, минорный, патч). В DaC подобную логику приходится реализовывать вручную и поддерживать силой командных соглашений.

Стандартный механизм Git для отметки релизов — это теги. Однако git tag — это просто указатель на конкретный коммит в истории. Он не содержит никакой семантической информации о том, что именно представляет собой этот релиз: «Руководство пользователя для версии продукта 3.5» или «Справочник API для релиза 1.2.1». Вся эта логика выносится в названия тегов (например, docs-v3.5) и требует строгой дисциплины и следованию установленным правилам [3].

Особенно остро эта проблема проявляется при необходимости поддерживать документацию для нескольких версий продукта одновременно. Представим ситуацию, когда компания поддерживает долгосрочную версию (LTS) 2.0, стабильную версию 3.0 и активно разрабатывает версию 4.0. Для каждой из них требуется свой, немного отличающийся набор документации. В Git это обычно решается созданием долгоживущих релизных веток (например, release/2.0, release/3.0). Если обнаруживается опечатка или ошибка, которую нужно исправить во всех трёх версиях, техническому писателю придётся вносить исправление в одну ветку, а затем аккуратно «переносить» (cherry-pick) этот коммит в другие. Этот процесс сложен, подвержен ошибкам и требует глубокого понимания Git. Более того, автоматизация публикации для нескольких версий становится нетривиальной задачей, требующей сложных скриптов и конфигураций в CI/CD, которые будут определять, какую ветку и куда публиковать. Специализированные платформы для документации часто имеют встроенные, удобные механизмы для управления такими версиями, чего полностью лишена стандартная VCS.

Переиспользование контента

Одной из самых мощных концепций в современном управлении технической документацией является компонентный подход (CCMS), предполагающий переиспользование контента [8]. Идея заключается в том, чтобы не копировать одинаковые фрагменты текста (например, предупреждения о безопасности, описания стандартных процедур, глоссарные статьи) из документа в документ, а определять их один раз и затем «включать» в нужные

места. Это гарантирует согласованность и значительно упрощает внесение изменений: достаточно поправить компонент в одном месте, и он автоматически обновится везде.

Системы контроля версий, работающие на уровне файлов, не поддерживают эту концепцию «из коробки». Для Git весь контент внутри файла — это единое целое. Он не умеет версионировать отдельный абзац или таблицу внутри большого документа. Если в одном файле содержится десять переиспользуемых фрагментов, и меняется только один из них, Git всё равно создаст новую версию всего файла. Это делает невозможным отслеживание истории изменений конкретного компонента.

На практике в экосистеме DaC эта проблема решается обходными путями. Чаще всего используется директива include (или аналогичная), которую предоставляют многие генераторы статических сайтов. Переиспользуемые фрагменты выносятся в отдельные небольшие файлы, которые затем включаются в основные документы на этапе сборки сайта. Такой подход работает, но имеет свои недостатки. Во-первых, логика переиспользования переносится из системы хранения в систему сборки, что делает её менее прозрачной. Во-вторых, это приводит к сильной фрагментации контента: проект может состоять из сотен или тысяч мелких файлов, что затрудняет навигацию и понимание общей структуры. Наконец, это не решает проблему условного контента — когда нужно, чтобы один и тот же фрагмент отображался по-разному для разных аудиторий или версий продукта. Реализация такой логики требует написания сложных скриптов или использования нестандартных расширений, что повышает хрупкость всей системы

Заключение

Все эти проблемы наглядно демонстрируют, что стандартные VCS имеют существенные функциональные пробелы, которые необходимо закрывать дополнительными инструментами (Рисунок 2).

Рисунок 2 – Функциональные пробелы стандартных VCS для задач документирования

Подход «Документация как код» предлагает мощную и современную альтернативу традиционным процессам. Однако его успешное внедрение невозможно без критического осмысления тех ограничений, которые несут в себе базовые инструменты этой парадигмы — системы контроля версий. Риск рассинхронизации с кодом, высокий инструментальный барьер для нетехнических специалистов, неудобство стандартных инструментов для рецензирования текста и сложности с управлением нетекстовыми файлами — это не досадные мелочи, а фундаментальные вызовы, которые могут свести на нет все потенциальные выгоды.

Слепое копирование практик разработки ПО без их адаптации к специфике создания документации одновременно и упрощает одни аспекты ведения проекта, и усложняет другие. DaC следует рассматривать не как готовое решение, а как основу для построения комплексной экосистемы. Успех зависит от того, насколько грамотно будут подобраны и интегрированы сопутствующие инструменты и процессы, целенаправленно решающие выявленные проблемы: системы автоматической проверки связей, удобные платформы для рецензирования, продуманные стратегии работы с бинарными файлами и, что самое важное, инвестиции в обучение и формирование новой культуры командной работы. Только такой взвешенный и комплексный подход позволяет в полной мере раскрыть потенциал DaC и построить по-настоящему эффективный и надёжный процесс создания технической документации.

Список источников:

1. Pro Git [Электронный ресурс] / S. Chacon, B. Straub. – 2014. – Режим доступа: https://git-scm.com/book/ru/v2 – Дата обращения: 19.05.2025.
2. Чакон С., Штрауб Б. Git для профессионального программиста / С. Чакон, Б. Штрауб; пер. с англ. И. Размайкина. – СПб.: Питер, 2024. – 494 с.
3. Clean Git History, или Тёмная сторона VCS [Электронный ресурс] // Хабр. – 2022. – Режим доступа: https://habr.com/ru/companies/ozontech/articles/754526/ – Дата обращения: 19.05.2025.
4. Макаровских Т. А. Документирование программного обеспечения. В помощь техническому писателю. – Екатеринбург: Издательские решения, 2022. – 212 с.
5. Бабич А. Делаем документацию здорового человека в Git на примере Docs Ozon [Электронный ресурс] // Хабр. – 2022. – Режим доступа: https://habr.com/ru/companies/ozontech/articles/695868/ – Дата обращения: 19.05.2025.
6. Как мы сократили время на проверку документов в IT с дней до часов [Электронный ресурс] // Хабр. – 2023. – Режим доступа: https://habr.com/ru/companies/T1Holding/articles/740488/ – Дата обращения: 19.05.2025.
7. Кузнецов Д., Певнева А. Создание системы документирования, или как мы от «ворда» к docs as code за месяц переходили [Электронный ресурс] // Хабр. – 2022. – Режим доступа: https://habr.com/ru/companies/cloud_ru/articles/686050/ – Дата обращения: 19.05.2025.
8. Как написать потрясающую техническую документацию? [Электронный ресурс] // Docsie. – 2021. – Режим доступа: https://www.docsie.io/blog/ru/articles/how-to-write-amazing-technical-documentation/ – Дата обращения: 19.05.2025.