**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**УКРАЇНСЬКА АКАДЕМІЯ ДРУКАРСТВА**

**КАФЕДРА ІНФОРМАЦІЙНОЇ, БІБЛІОТЕЧНОЇ ТА АРХІВНОЇ СПРАВИ**

РОБОТА ПЕРЕВІРЕНА І ДОПУЩЕНА ДО ЗАХИСТУ

Зав. кафедри ІБАС та ІМ,

проф., д-р іст. наук

Михайло ПАСИЧНИК.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (підпис)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (дата)

**КВАЛІФІКАЦІЙНА РОБОТА**

на здобуття освітнього ступеня магістра зі спеціальності

029 «Інформаційна, бібліотечна та архівна справа»

за магістерською програмою професійного спрямування

«Документознавство та інформаційна діяльність» на тему:

«**Використання систем контролю версій для колективної роботи над мультимедійною документацією**»

Виконав:

студент групи ДКМ-61

Андрєєв Андрій Олексійович  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (підпис)

Науковий керівник:

кандидат філологічних наук, доцент

Ткач Леся Миколаївна  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (підпис)

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань. Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (підпис)

ЛЬВІВ – 2023

# ЗМІСТ

[**ЗМІСТ 2**](#_m07p567sm1hl)

[**ВСТУП 4**](#_26zbzsr42tc)

[Мета і завдання дослідження 4](#_qpnog1wyjqhf)

[Об’єкт дослідження 5](#_2wkreipu1kxi)

[Предмет дослідження 5](#_2a6ljs42nbob)

[Наукова новизна одержаних результатів 5](#_y3l3o6297d4l)

[**Розділ 1 6**](#_vzy0aw3n228d)

[**Версії 6**](#_w9ct4t9kui83)

[Наука 6](#_33ro2541o3k2)

[Мистецтво 7](#_btvhh4idyxgk)

[Юридична і законотворча діяльність 8](#_hbcznkwlfqne)

[Переклади 8](#_6bdh5y4r2k6d)

[Індустрія та інше 10](#_ypk64cv2u72e)

[**Розділ 2 13**](#_y0eoobhe43fg)

[**Колективна робота 13**](#_175dbn7394f9)

[Процеси та практики 14](#_knqpaenstjxx)

[Взаємні ревьʼю 15](#_dw9qn1gphhxk)

[Інструменти 16](#_1u31edtnaiaz)

[Інші чинники 18](#_yc35xpu64dbz)

[Різноманітність 18](#_tl46jc76kz2v)

[Локалізація та інтернаціоналізація 19](#_g8fijge9o3hf)

[Цільова аудиторія 21](#_8gl3qkwpinwu)

[**Розділ 3 23**](#_gneucxwbxto)

[**Теоретичні основи систем контролю версій 24**](#_7u29sv1ot1rs)

[Історія розвитку систем контролю версій. 24](#_bejmzisi4pgw)

[Історичний контекст систем контролю версій. 24](#_zdbnpvf2wmmb)

[Ручні методи контролю версій 25](#_6wpja456ckqx)

[Систематичний конроль 27](#_1hn9pc3y9yyi)

[Перше покоління: локальні системи контролю версій 28](#_xrxv4iag2c3l)

[Друге покоління: централізовані системи контролю версій 29](#_dskyof7y71yy)

[Третє покоління: розподілені системи контролю версій 31](#_r0fu6van09pq)

[Сучасні тенденції та перспективи на майбутнє 33](#_t52lmkfldp4k)

[Основні терміни 35](#_vlm3pejd089j)

[Основні поняття 42](#_jker40vw2bi0)

[Основні принципи роботи систем контролю версій 44](#_g2ntqf8mxrak)

[Робочі процеси в системах контролю версій 45](#_c6ckmu42qq62)

[Інтеграція з сучасними практиками розробки 48](#_tve3pea09l2e)

[Класифікація систем контролю версій 51](#_axifmk2uhsod)

[За архітектурою 51](#_df0isncnrde9)

[За функціональністю та характеристиками 52](#_uixwt273betp)

[За методом доступу 54](#_l91dsnrrro54)

[За рівнем інтеграції 56](#_xche7dvw808)

[За галузевою специфікою використання 59](#_af5uyww295k9)

[Еволюційні тенденції в класифікації 61](#_i3w8dxanmz3s)

[**Список використаних джерел 63**](#_tecnuaah1mr)

# ВСТУП

Сучасні програмні та апаратні продукти є складними системами, що потребують ґрунтовної мультимедійної документації. Розробка такої документації, як і самих продуктів, є ітеративним колективним процесом. Ефективна спільна робота над документацією вимагає побудови процесів для взаємодії співавторів і кінцевих споживачів, своєчасного внесення і публікації змін та отримання зворотного звʼязку, а також використання спеціальних технічних рішень для перегляду, порівняння, редагування та узгодження змін у документах. Одним з таких рішень є системи контролю версій, зокрема Git та GitHub, які набули широкого поширення останнім часом. В цій роботі доводиться доцільність використання систем контролю версій для колективної роботи над документацією.

Вивчення можливостей цих систем для створення колективної мультимедійної документації є актуальним, оскільки дозволяє поліпшити якість технічної документації, прискорити її розробку, оптимізувати взаємодію у командах авторів та розробників. Крім того, поширення використання таких систем контролю версій сприятиме підвищенню кваліфікації фахівців в Україні згідно із світовими стандартами.

Отже, дослідження є актуальним і має як теоретичне, так і практичне значення.

## Мета і завдання дослідження

Мета роботи - обґрунтувати доцільність використання систем контролю версій Git та GitHub для колективного створення і супроводження мультимедійної технічної документації.

Для досягнення мети необхідно виконати такі завдання:

* Обґрунтувати доцільність збереження версій та контролю історії змін.
* Виявити практики і інструменти, які збільшують ефективність колективної роботи над документами.
* Проаналізувати теоретичні основи та історію виникнення систем контролю версій.
* Опанувати базову термінологію Git та особливості роботи в GitHub.
* Розробити приклад мультимедійної технічної документації з використанням можливостей GitHub.

## Об’єкт дослідження

Об’єктом дослідження є процес створення і супроводження колективної мультимедійної технічної документації.

## Предмет дослідження

Предметом дослідження є можливості систем контролю версій Git та GitHub для колективної розробки, зберігання та оновлення мультимедійної технічної документації.

## Наукова новизна одержаних результатів

Наукова новизна роботи полягає в тому, що:

* Комплексно досліджено використання систем контролю версій Git та GitHub для створення і супроводження колективної мультимедійної технічної документації.
* Дістали подальший розвиток підходи до організації колективної розробки технічної документації на основі Git workflow.
* Запропоновано рекомендації щодо оптимального використання функціоналу GitHub для технічних письменників.

# Розділ 1

# Версії

Факт існування будь-якого документа означає існування багатьох його версій, адже створення документів - це поступовий, ітеративний процес. Між першим чорновиком і готовністю до публікації будь-який документ проходить проміжні етапи, і на кожному з них вміст документа поступово змінюється. Якщо недоцільно зберігати проміжні результати - створені таким чином неявні версії документів існують короткий час як ефемерні сутності. Якщо етап зміни документа логічно завершений, такий, що вважається значущим або важливим - доцільно зафіксувати його стан явно, таким чином створворивши персистентну версію документа. Пригадаємо випадки, коли існування окремих версій буде доцільне, корисне та необхідне.

## Наука

* **Академічні публікації**. Перед публікацією, наукові статті та книги проходять через багато етапів редагування та коректури. Саме зберігання всіх версій допомагає відслідковувати зміни та вносити необхідні виправлення.
* **Розробка навчальних матеріалів**. Під час створення навчальних матеріалів, викладачі і методисти можуть вносити зміни, доповнення та корекції. Існування різних версій дозволяє адаптувати навчальні матеріали та обрати оптимальний варіант для навчального процесу з урахуванням методики навчання, а також віку, фаху та освітнього рівня цільової аудиторії.
* **Дослідження та експерименти.** Під час проведення наукових досліджень, кожен етап експерименту вимагає фіксації. Саме зберігання кожної версії дозволяє науковцям досліджувати тривалі процеси, а в подальшому - систематизувати знання та результати досліджень.
* **Історичні свідоцтва і хроніки**. Фіксація подій, що відбувалися в минулому, здійснена різними дослідниками або протягом різних часових інтервалів критично важлива для всебічного ретроспективного аналізу історичних подій та джерел.

## Мистецтво

* **Музика**. Саме існування різних версій музичних творів, варіантів аранжувань опер, варіантів виконання симфоній, кавер-версії популярних пісень і робить музичне мистецтво різноманітним. Транспозиція в іншу тональність дозволяє співати пісні виконавцям із різним голосовим діапазоном.
* **Кіно і театр**. Ремейки кінострічок, різні постановки пʼєс і вистав. Режисери та актори можуть інтерпретувати п'єси по-різному, зміщувати акценти, по-різному висловлювати емоції, інакше виражати взаємодію людей із навколишнім світом.
* **Балет і танець**. Режисери і хореографи можуть створювати різні версії вистав і танців на одну і ту ж музику. Танцюристи можуть виконувати одну і ту ж хореографію по-різному.
* **Література**. Дитячі адаптації літературних творів роблять складні твори більш доступними та зрозумілими для дітей, враховуючи їхній вік та рівень розвитку, стимулюють дитячий інтерес до читання. Перетворення літературних творів на аудіокниги або комікси чи візуальні новели створює нові жанри і напрямки мистецтва.
* **Скульптура і живопис**. Процес створення реплік відомих творів є необхідним етапом навчання митців. І навіть при створенні копій, експерименти із різними матеріалами та техніками роблять кожну копію унікальним витвором мистецтва.
* **Фотографія**. Ретуш та обробка. Одне фото може мати безліч версій, залежно від того, як його оброблено та ретушовано. Фотограф може також робити кілька знімків одного і того ж об'єкта з різних ракурсів та при різному освітленні.
* **Відеоігри**. Порти, ремейки та модифікації. Консольні версії відрізняються від версій для персональних компʼютерів, можуть вимагати адаптації управління і ігрових механік. Спільноти гравців та розробники можуть створювати модифікації для ігор, змінюючи графіку, геймплей, ігровий наратив тощо.

## Юридична і законотворча діяльність

* **Ревізії документів та договорів**. Юридичні документи часто підлягають численним ревізіям та внесенню змін, і збереження кожної версії є критично важливим для відстеження змін та домовленостей, укладених у різний час. Наприклад в закон України про правила дорожнього руху [1] в період з 2005 по 2023 роки було внесено 43 правки згідно з постановами Кабінету Міністрів України і 3 правки згідно з додатковими судовими постановами.
* **Етапи переговорів**. Під час переговорів створюється кілька версій договорів, які відображають еволюцію консенсусу між сторонами.
* **Законодавство**. Законодавчі документи проходять кілька ревізій та внесення змін до та після того, як вони були прийняті, причому кожна версія має як юридичне так і історичне значення. Проектні версії, створені для консультацій та відгуків, можуть відрізнятися від початкових проектів та кінцевих прийнятих версій.
* **Правозастосування**. Для конкретних випадків правозастосування критично важливим є саме відповідність життєвого випадку конкретній версії закону або юридичної норми. Одна і та сама подія може бути кваліфікована як легальна або нелегальна в залежності від того, які версії законів були чинні, коли відбулася ця подія.

## Переклади

Переклад відіграє ключову роль у спілкуванні та взаєморозумінні між людьми з різних мовних та культурних середовищ. Передача змісту з однієї мови в іншу - складний творчий процес, який означає не тільки і не стільки механічну заміну слів за словником, але має враховувати культурний, соціальний, історичний контекст для повної і адекватної передачі інформації, ідей та емоцій. При цьому переклад завжди створює нову версію, яка буде радикально відрізнятись від оригіналу, адже в ній може не бути жодної спільної з оригіналом літери. Цікаво, що перекладений твір завжди ототожнюється з оригіналом, і автором перекладеного твору все одно вважається не перекладач, який написав весь текст перекладу, а саме автор оригіналу, хоча він не написав жодного слова перекладу.

Існування декількох версій одного тексту різними мовами дозволяє не тільки доносити інформацію до носіїв різних мов і вивчати нові мови на момент написання тексту, але й дає можливість через тисячі років (!) відновлювати втрачені, мертві мови. Один з найвідоміших прикладів - Розетський камінь [2]. Стела, на якій 196 року до н. е. було висічено три версії указу мемфіських жерців. Давньоєгипетським скорописом, давньоєгипетськими ієрогліфами, а також давньогрецькою. В ХІХ ст саме існування трьох версій тексту, написаних різними мовами, зробило можливим розшифровку єгипетських ієрогліфів Жаном-Франсуа Шампольйоном і іншими дослідниками.

Для якумога точнішої передачі змісту іноді просто було недостатньо однієї мови. Відомий приклад багатомовних версій тексту Біблії - це Поліглотти [3], видання Біблії декількома мовами, що були створені з метою кращого розуміння текстів Святого Письма. Поліглотти містили текст написаний двома або більше (найбільш повні 10 - 12) мовами, зазвичай у паралельних колонках або на сусідніх сторінках. Використовувались такі видання як в релігійних, теологічних цілях, так і для академічних, філологічних досліджень.

Таким чином, існування різних версій (перекладів) дозволяє:

* Подолати лінгвістичні барʼєри при спілкуванні
* Вивчати іноземні мови
* Оживляти мертві мови
* Точніше передавати зміст

## Індустрія та інше

Існування різних версії документів - невідʼємна частина будь-якої індустрії, адже саме можливість створювати нові версії дозволяє адаптувати виробничі, економічні та інші процеси відповідно до оточуючої реальності. Загальновідомі приклади такі:

**Стандарти**. Організації, що розробляють стандарти (наприклад, ISO або ДСТУ) випускають нові версії стандартів, щоб відображати технологічний прогрес або зміни в галузі. Зазвичай, нова версія стандарта містить рік оновлення в назві своєї версії. Наприклад державний стандарт *ДСТУ 7564:2014 Інформаційні технології. Криптографічний захист інформації. Функція хешування* [4] заміщує застарілий *ГОСТ 34.311-95 Інформаційна технологія. Криптографічний захист інформації. Функція хешування*. А міжнародний стандарт *ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models* [5] прийшов на заміну *ISO/IEC 9126 Software engineering — Product quality.*

Застосування нових версій стандартів необхідне, оскільки вони враховують найновіші досягнення, дослідження та практики в галузі. Із часом технології, методики та підходи до роботи змінюються, і стандарти мають відображати ці зміни, щоб залишатися актуальними та корисними. У випадку функцій хешування, наприклад, нові стандарти забезпечують криптографічну стійкість цифрового підпису. У випадку стандартів, що стосується розробки програмного забезпечення - додаються нові метрики для оцінки якості.

Застосування нових стандартів допомагає організаціям підтримувати високий рівень якості своєї продукції чи послуг, дбати про безпеку, ефективність та надійність. Крім того, вони сприяють гармонізації та інтеграції різних систем, продуктів або послуг на міжнародному рівні. Таким чином, організації, які впроваджують нові, актуальні версії стандартів:

* Можуть бути впевнені, що їх продукція або послуги відповідають найсучаснішим вимогам ринку.
* Підвищують довіру споживачів, які розуміють, що продукт або послуга відповідає визнаним стандартам якості.
* Спрощують процес взаємодії з партнерами, клієнтами або регулюючими органами.

**Медицина**. У випадку, коли для адаптації до змін потрібно враховувати не лише технологічний поступ, але й вимоги регулюючих органів, формалізація процесу створення нових версій документів і збереження історії змін є особливо важливі. Нові версії документів ВООЗ, що містять оновлені рекомендації протоколів лікування спираються на нові наукові дослідження та оновлені результати систематизацій попередніх.

Характерним прикладом щорічної зміни рекомендацій Всесвітньої організації охорони здоровʼя - є список штамів грипу, який актуальний лише один вакцинальний сезон [6]. Віруси грипу мутують настільки швидко, що застосування застарілих вакцин робить процесс вакцинації неефективним. Застосування ж нових версій вакцин, навпаки - забезпечує ефективність профілактики захворювання.

Інший приклад - це оновлені рекомендації для для лікування гіпертензії у дорослих, які містять зміни щодо порогових значень для ініціювання лікування, інтервалів моніторингу, цільового артеріального тиску та можливості призначення лікування не лікарями [7]. Важливість правильного лікування і своєчасного оновлення відповідних рекомендацій у випадку гіпертензії важко переоцінити, оскільки хвороби системи кровообігу за статистикою були і залишаються основною причиною смерті не тільки в Україні [8], але в усьому світі [9].

**Програмне забезпечення**. Існування різних версій програмного забезпечення в індустрії розробки має декілька корисних аспектів:

1. **Безпека**: Коли виявляється вразливість в програмному забезпеченні, розробники можуть випустити нову версію, яка виправляє цю вразливість. Користувачі, які оновлюють своє ПЗ до останньої версії, отримують захист від потенційних загроз.
2. **Нові функції та покращення**: З кожною новою версією програмного забезпечення можуть бути додані нові функції, які роблять продукт більш потужним або зручним для користувача.
3. **Виправлення помилок**: Помилки та баги, які були в попередніх версіях, можуть бути виправлені в нових версіях.
4. **Сумісність**: Різні версії програмного забезпечення можуть бути сумісні з різними версіями операційних систем або іншого ПЗ. Це дозволяє користувачам, які ще не готові або не можуть оновити свою ОС або інше ПЗ, продовжувати використовувати старі версії програмного забезпечення.
5. **Тестування та стабільність**: Іноді організації або розробники можуть випустити "бета" або "альфа" версії ПЗ для тестування. Це дозволяє отримати відгук від користувачів перед тим, як випустити офіційну версію.
6. **Версії для різних ринків**: Розробники можуть випускати різні версії ПЗ для різних ринків або груп користувачів. Наприклад, може існувати базова версія для звичайних користувачів та розширена версія для професіоналів.
7. **Відновлення після проблем**: Якщо нова версія ПЗ містить критичні помилки або проблеми, користувачі можуть повернутися до попередньої стабільної версії.
8. **Версійний контроль**: Існування різних версій дозволяє розробникам вести документацію та контроль змін, що полегшує процес розробки та співпраці.

Всі ці аспекти роблять існування різних версій важливим елементом в індустрії розробки програмного забезпечення.

Підсумовуючи, на основі наведеного аналізу можна зробити висновок, що існування множинних версій документів є невід'ємною складовою різноманітних сфер людської діяльності. В науковій сфері фіксація проміжних результатів досліджень уможливлює ретроспективний аналіз еволюції знань. В художній творчості створення альтернативних інтерпретацій сприяє новаторству та експериментам. В юриспруденції версіювання законодавчих актів забезпечує правильність правозастосування. У сфері перекладу наявність кількох мовних версій робить можливим подолання комунікативних бар'єрів. В індустріальній сфері оновлення стандартів та програмного забезпечення сприяє підвищенню безпеки, ефективності та сумісності. Тким чином, можна стверджувати, що версіювання документів є критично важливим чинником науково-технічного прогресу та культурної еволюції.

# Розділ 2

# Колективна робота

Важливість колективної роботи очевидна для виробничих галузей, адже група людей фізично може зробити більше, ніж одна людина. Також очевидно, що не всі види діяльності масштабуються ідеально, адже девʼять жінок не народять дитину за місяц, а тисяча школярів не здатні виконувати функцію одного викладача. Тому особливо цікавими є дослідження колективної роботи в контексті інтелектуальної діяльності. В дослідженні на тему "Зростаюче домінування команд у виробництві знань" [10] автори проаналізували майже 20 мільйонів наукових робіт та патентів за 5 десятиліть, щоб вивчити тенденції в індивідуальних та колективних дослідженнях. Вони виявили такі факти:

* У всіх основних галузях науки та техніки, суспільних наук, мистецтва, гуманітарних наук та патентів спостерігається стійка тенденція переходу від індивідуальних до колективних досліджень.
* У природничих науках відсоток статей, опублікованих індивідуальними авторами, знизився з близько 85% у 1950-х роках до 45% у 2000 році. Подібні тенденції спостерігалися у суспільних науках та мистецтві та гуманітарних науках.
* Наразі дослідницькі колективи випускають роботи з найбільшим впливом, навіть у галузях, де раніше домінували індивідуальні автори. Колективи частіше створюють високоцитовані дослідження, ніж окремі автори.
* На початкових етапах, проаналізованих у дослідженні, окремі автори частіше створювали найбільш видатні високоцитовані роботи в науці та техніці та суспільних науках. Але вже в 2000 році роботи, написані колективами, домінували у верхній частині індексів цитувань.
* Зростаючі масштаби, складність та вартість сучасних досліджень, ймовірно, пояснюють збільшення кількості колективних робіт в лабораторних дослідженнях. Але подібне зростання кількості колективних робіт в математиці, економіці тощо свідчить про додаткові фактори, такі як зростання кількості дослідників, спеціалізацію та зниження витрат на комунікацію.

Ефективна колективна робота над документацією, як технічною, так і нетехнічною, вимагає комбінації процесів, практик та інструментів, розроблених для сприяння комунікації, координації та обміну інформацією між членами команди. Перелічимо основні складові успішних стратегій.

## Процеси та практики

**Стандарти.** Встановлення стандартів документації є важливим для ефективної колективної роботи. У разі регульованих індустрій, таких як телекомунікації, або фінанси, наявність стандартів - критично важлива. Детальний стандарт забезпечує узгодженість та чіткість у всіх документах, визначає вимоги до створення і підтримки документації, в також враховує фактори, актуальні для ефективної роботи над документацією. Наприклад міжнародний стандарт ISO/IEC/IEEE 26511:2018 [11] містить такі аспекти системного та програмного інжинірингу як вимоги для замовників та постачальників користувацької документації:

* Описує вимоги та процедури для управління інформацією для користувачів протягом життєвого циклу розробки програмного забезпечення, інформаційних послуг та систем.
* Застосовується до організацій та осіб, що створюють набори даних, окремі інформаційні проекти розробки та внутрішню або службову документацію.
* Охоплює діяльність з управління інформацією, таку як запуск проекту, формування команди, управління інформацією через переклад/локалізацію, вимірювання продуктивності та якості тощо.
* Визначає процеси стратегічного планування, планування проектів, оцінки потреб користувачів, формування команд, управління проектами, вимірювання якості та продуктивності тощо.
* Визначає вимоги до ключових документів, таких як стратегічні плани та плани проектів.
* Не залежить від конкретних програмних засобів і застосовується до друкованої і електронної форм документації.
* Є актуальним для менеджерів, розробників та замовників інформаційних систем, рецензентів, тестувальників та інших, хто залучений до розробки інформаційних систем.
* Мета стандарту - побудова процесів, що дозволяють випускати високоякісну, актуальну, точну документацію для користувачів рентабельним способом.

### **Взаємні ревьʼю**

Регулярний взаємний перегляд документів колегами, співавторами та рецензентами і надання зворотного зв'язку забезпечує високу якість документації [12]. Мета взаємного рецензування - це надати можливість іншому співавтору або рецензенту переглянути ваш документ та визначити сфери для його вдосконалення. Взаємне ревьʼю допомагає покращити якість документа шляхом отримання об'єктивного погляду з боку сторонньої особи, залученню іншої точки зору. Цей процес конструктивний, коли співавтори мають спільну ціль - створити якісний документ і надають один одному регулярний зворотний зв'язок для покращення загальної якості документів [13]. Ефективний процес взаємного рецензування надає такі переваги:

* Вирішує основні питання структури, мови, слідування загальним правилам та шаблонам
* Допомагає узгодити стилі написання документів в команді
* Досвідченні рецензенти можуть приносити цінні ідеї
* Забезпечує технічну точність з мінімальними зусиллями від оригінального автора

### **Інструменти**

**Платформи для спільної роботи над документами**. Інструменти як Wiki, Google Docs, Microsoft Office Online або Confluence дозволяють багатьом користувачам разом працювати над документами, відстежувати зміни та залишати коментарі. В залежності від того, як організований робочий процес, спільна робота над документом може бути одночасною, серіалізованою або повністю асинхронною. Кожен варіант має свої переваги і недоліки [14].

* Одночасна. Всі співавтори мають можливість вносити зміни одночасно.
  + *Переваги*: Швидка взаємодія між учасниками, можливість відразу бачити зміни, які вносять інші, та відповідно реагувати на них. Це особливо корисно для оперативних зустрічей або брейнштормів.
  + *Недоліки*: Можливі конфлікти змін, якщо два користувача намагаються вносити зміни в одну і ту ж частину документа. Також потребує постійного підключення до Інтернету.
* Серіалізована. Співавтори вносять зміни по черзі, тільки один автор може змінювати документ, всі решта чекають, поки він закінчить.
  + *Переваги*: Зменшує ризик конфліктів змін, оскільки тільки одна особа може редагувати документ у певний момент часу. Це може бути корисно для документів, які потребують послідовного огляду або редагування.
  + *Недоліки*: Процес редагування може бути повільнішим, оскільки інші учасники мають чекати, поки їхній черга прийде. Це може призвести до затримок у роботі.
* Асинхронна. Співавтори працюють паралельно, незалежно один від одного, до тих пір, поки не захочуть обʼєднати свої правки.
  + *Переваги*: Гнучкість у роботі, дозволяє учасникам працювати в зручний для них час без необхідності чекати на інших. Відмінно підходить для команд, розташованих в різних часових поясах.
  + *Недоліки*: Можливі конфлікти при об'єднанні змін, якщо два або більше учасників вносили зміни в одні й ті же частини документа. Потребує додаткового часу на об'єднання та узгодження змін.

**Системи контролю версій**. Системи як Perforce, Subversion, Mercurial, Git дозволяють командам керувати змінами та зберігати всі версій документів, що особливо корисно для технічної документації. Розрізняють такі види систем контролю версій:

**Локальні** системи контролю версій:

Ці системи працюють на одному комп'ютері та відстежують зміни у локальних файлах. Приклад: RCS (Revision Control System [15]). Локальна система контролю версій - це локальна база даних, розташована на вашому локальному комп'ютері, в якій кожна зміна файлу зберігається як окрема дельта від попередньої версії. Щоб побачити, як виглядав файл у певний момент, необхідно виконати всі зміни до файлу по порядку від самої першої його версії до цього моменту.

Основною проблемою є те, що все зберігається локально. Якщо щось трапиться з локальною базою даних, всі патчі будуть втрачені. Якщо щось трапиться з якоюсь окремою версією, всі зміни, зроблені після цієї версії, будуть втрачені. Також дуже важко або майже неможливо співпрацювати з іншими розробниками або командою.

**Централізовані** системи контролю версій:

Ці системи використовують один централізований сервер, на якому зберігаються всі версії файлів, і клієнти, які отримують копії файлів з цього сервера. Приклади: SVN (Subversion [16]), CVS (Concurrent Versions System [17]), Perforce [18]. Централізована система контролю версій має єдиний сервер, який містить всі версії файлів. Це дозволяє багатьом клієнтам одночасно отримувати доступ до файлів на сервері, завантажувати їх на свій локальний комп'ютер або відправляти їх на сервер зі свого локального комп'ютера. Адміністратори контролюють, хто що може робити. Це дозволяє співпрацювати з іншими розробниками або командою.

Принциповою проблемою цієї структури є те, що все зберігається на централізованому сервері. Якщо щось трапиться з цим сервером, ніхто не зможе зберегти свої версіоновані зміни, витягнути файли або взагалі співпрацювати. Як і у випадку з локальним контролем версій, якщо центральна база даних пошкодиться, і резервні копії не зберігалися, втрачається вся історія проекту, крім окремих файлів, які випадково збереглись у людей на локальних машинах.

**Розподілені** системи контролю версій:

У таких системах кожен користувач має повну копію репозиторію, тобто він має всю історію змін. Приклади: Git [19], Mercurial [20], Bazaar [21], Darcs [22]. З розподіленими системами контролю версій клієнти не просто отримують останній знімок файлів з сервера, вони повністю дублюють сховище, включаючи всю його історію. Таким чином, кожен, хто співпрацює над проектом, володіє локальною копією всього проекту, тобто володіє власною локальною базою даних з власною повною історією.

За такої моделі, якщо сервер стає недоступним або виходить з ладу, будь-яке з клієнтських сховищ може надіслати копію версії проекту будь-якому іншому клієнту або знову на сервер, коли він стане доступний. Достатньо, щоб у одного клієнта була правильна копія, яку потім можна легко розповсюдити далі. Git - найвідоміший приклад розподілених систем контролю версій.

Будь-яка система контролю версій дає можливість відслідковувати і контролювати зміни, і дає суттєві переваги в роботі порівняно з випадком, коли контролю версій немає взагалі. Але через недоліки локальних і централізованих систем, в сучасній практиці переважно використовують розподілені системи контролю версій.

## Інші чинники

### **Різноманітність**

Існує багато досліджень, які показують переваги різноманітності в науковій роботі. Наприклад в роботі на тему “Перевага етнічної різноманітності у науковій співпраці” [23] де проаналізовано понад 9 мільйонів наукових робіт та 6 мільйонів вчених, щоб дослідити зв'язок між науковим впливом досліджень та різними типами різноманітності, виявлено наступне:

* Етнічна різноманітність у дослідницьких командах має найсильніший зв'язок з науковим впливом, вимірюваним кількістю цитувань. Це було несподіваним висновком, оскільки етнічна приналежність, як до того вважалося, менш пов'язана з технічною компетентністю, ніж такі фактори, як академічна дисципліна.
* Використовуючи випадкові базові моделі та укрупнене точне співставлення, дослідники виявили переконливі докази того, що роботи та вчені з більшою етнічною різноманітністю перевершили менш різноманітних колег. Для високоцитованих робіт і вчених етнічна різноманітність призвела до зростання впливу більш ніж на 10% і 47% відповідно.
* Результати припускають, що рекрутери повинні прагнути заохочувати етнічну різноманітність у дослідницьких командах, оскільки, схоже, це дає реальні переваги, такі як поліпшення продуктивності та впливу, ймовірно, через відмінності в поглядах і культурі.
* Результати підкреслюють, що крім технічної компетентності, при формуванні різноманітних дослідницьких груп слід враховувати такі фактори, як згуртованість команди та почуття єдності.

Зауважимо, що для побудови дійсно різноманітної команди в сучасних умовах доводиться організовувати взаємодію людей в різних часових поясах. Таким чином, доцільною є побудова **асинхронних** процесів робочої комунікацій за допомогою **розподілених** систем, таких як Git.

### **Локалізація та інтернаціоналізація**

В сучасних умовах різноманітною буде також цільова аудиторія. Всі сучасні інформаційні системи мають враховувати необхідність локалізації і інтернаціоналізації [24]. Визначення цих термінів, які дає Консорціум Всесвітньої Мережі (W3C) - наступні [25].

**Локалізація** - це адаптація продукту, додатку чи вмісту документа для задоволення мовних, культурних та інших вимог конкретного цільового ринку (локалі). Часто локалізація помилково сприймається лише як синонім перекладу інтерфейсу користувача та документації, але насправді локалізація є набагато складнішим процесом. Вона може включати в себе пристосування, пов'язані з:

* Числовими форматами,
* Часовими форматами і форматами дат в календарі
* Використанням валюти
* Використанням клавіатури
* Алфавітним сортуванням
* Символами, піктограмами та кольорами
* Текстом та графікою, що містять посилання на об'єкти, дії чи ідеї, які в певній культурі можуть бути неправильно зрозумілі чи неприйнятні
* Юридичними вимогами

Локалізація іноді може вимагати повного переосмислення логіки візуального дизайну, якщо спосіб ведення бізнесу (наприклад, бухгалтерський облік) або прийнята парадигма (наприклад, фокус на окремій особі або групі) у даній місцевості або цільовій аудиторії істотно відрізняється від вихідної культури.

**Інтернаціоналізація** - це таке проектування та розробка продукту, додатку або вмісту документа, яке дозволяє легку подальшу локалізацію для різних цільових аудиторій, які відрізняються культурою, регіоном або мовою.

Інтернаціоналізація, як правило, передбачає:

* Проектування та розробку таким чином, щоб усунути бар'єри для локалізації або міжнародної дистрибуції. Одне з основних вимог при цьому - використання Unicode або забезпечення належної обробки застарілих варіантів кодувань символів, уникнення залежності коду від значень рядків інтерфейсу користувача тощо.
* Забезпечення підтримки функцій, які можуть не використовуватися до тих пір, поки не відбудеться локалізація. Наприклад, додавання розмітки для підтримки двонаправленого тексту або для ідентифікації мови.
* Підтримка локальних, регіональних, мовних уподобань. Зазвичай це включає в себе вбудовані дані і функції локалізації, отримані з існуючих бібліотек або користувальницьких налаштувань. Приклади включають формати дати і часу, локальні календарі, числові формати і числові системи, сортування та подання списків, поводження з особистими іменами та формами звернення тощо.
* Відокремлення локалізованих елементів від вихідного коду або вмісту, так що локалізовані альтернативи можна завантажувати або вибирати за потреби на основі міжнародних уподобань користувача.

Варто памʼятати, що ці пункти не обов'язково включають локалізацію вмісту, додатку чи продукту на іншу мову. Це практики проектування і розробки, які дозволяють легко здійснити таку міграцію в майбутньому. При цьому вони також корисні навіть якщо локалізація ніколи не відбудеться. Ідеально, щоб інтернаціоналізація відбувалася як фундаментальний крок у процесі проектування та розробки, а не як додаткове міркування яке відбулося постфактум і викликало потребу в повному перепроектуванні.

Як бачимо із визначень, інтернаціоналізація і локалізація вимагатимуть використання процесів, повʼязаних із порівнянням рядків тексту, різних його версій. Для збереження і порівняння версій великої кількості тексту у промислових масштабах використання системи контролю версій буде чинником, що критично впливає на результат. Використання саме розподіленої системи контролю версій при цьому дозволить асинхронно узгоджувати роботу людей з різних країн та часових зон.

### **Цільова аудиторія**

Крім колективу, що безпосередньо працює над документацією, на якість документів також впливає і цільова аудиторія [26]. Ефективний процес підтримки належної якості документації має включати:

* Тестування документації. Залучення внутрішніх та зовнішніх груп для перевірки документації допомагає виявити недоліки, незрозумілості або неточності [27]. Внутрішні групи можуть включати інших членів команди, тоді як зовнішні групи можуть бути представниками цільової аудиторії або незалежними експертами.
* Своєчасні оновлення документації. Використання систем неперервної інтеграції та неперервної доставки (CI/CD) дозволяє автоматизувати процес оновлення документації, гарантуючи, що вона завжди актуальна і відповідає останнім змінам у продукті чи сервісі.
* Інтеграція з іншими системами. Інтеграція документації з пошуковими системами та соціальними мережами підвищує її видимість та доступність для користувачів [28]. Це також може сприяти зворотньому зв'язку та зміцненню взаємодії з користувачами.
* Отримання зворотного звʼязку. Вкрай необхідне для ефективної підтримки належного рівня якості документації. В сучасних умовах існує багато інструментів і продуктів, що максимально спрощують взаємодію з кінцевими споживачами. Пошукові системи, агрегатори, wiki ресурси, спеціалізовані форуми, групи в соціальних мережах, тощо.

Зауважимо, що використання розподілених систем контролю версій, таких як Git, та продуктів як GitHub, дозволяє залучати спільноту користувачів до процесу створення та вдосконалення документації. За допомогою механізму пулл-реквестів користувачі можуть пропонувати виправлення, а за допомогою функцій опису проблем (Issues) – повідомляти про проблеми або обговорювати нові функції

Підсумовуючи надану інформацію, можемо зробити такі проміжні висновки:

* Дослідження показують зростаючу тенденцію переходу від індивідуальних до колективних досліджень у всіх галузях науки і техніки. Колективи частіше створюють високоцитовані дослідження, ніж окремі автори.
* Ефективна колективна робота вимагає належних процесів, практик та інструментів. Серед них - стандарти документації, взаємні рев'ю, спільні платформи, системи контролю версій.
* Розподілені системи контролю версій, на відміну від локальних і централізованих, забезпечують надійне збереження всієї історії змін. Це дозволяє ефективну співпрацю та асинхронну координацію колективної роботи.
* Системи контролю версій є важливою складовою процесів локалізації та інтернаціоналізації, оскільки дозволяють зберігати та порівнювати різні версії перекладів.
* Використання систем на кшталт GitHub дозволяє залучати спільноту користувачів до процесу удосконалення документації за допомогою механізмів пулл-реквестів та повідомлення про помилки.

Отже, використання розподілених систем контролю версій є важливим чинником підвищення ефективності та якості колективної роботи над документацією в сучасних умовах.

# Розділ 3

# Теоретичні основи систем контролю версій

## Історія розвитку систем контролю версій.

### Історичний контекст систем контролю версій.

Еволюція систем контролю версій (СКВ) є процесом, що віддзеркалює стрімкий розвиток комп'ютерних технологій та зростаючу складність розробки програмного забезпечення. Щоб краще зрозуміти значення систем контролю версій у сфері програмної інженерії та за її межами, важливо зануритися в їхній історичний контекст, простеживши шлях до ери, коли керування різними версіями програмного забезпечення та документів було ручним і громіздким процесом.

У найперші дні програмування комп'ютерів, протягом 1950-х і 1960-х років, концепція контролю версій практично не існувала. Програмісти писали код на перфокартах і змушені були вручну керувати версіями свого програмного забезпечення. Цей процес був неефективний, часто призводив до плутанини та втрати даних. По мірі зростання розміру та складності програмних проектів, потреба у систематичному способі керування змінами стала очевидною.

1970-ті роки ознаменувалися початком більш структурованого, хоча ще й ручного, підходу до розробки програмного забезпечення. Програмісти почали використовувати методики, такі як зберігання коду в декількох каталогах або використання конвенцій іменування файлів для розрізнення версій [29]. Однак ці методи були недосконалими і не вирішували зростаючих проблем у спільній командній розробці та відстеженні версій.

Концепція контролю версій почала набувати більш конкретної форми з появою системи контролю вихідного коду (Source Code Control System, SCCS) у 1972 році, розробленої Марком Дж. Рочкіндом у Bell Labs. SCCS була першою системою, яка автоматизувала завдання відстеження різних версій файлів вихідного коду, запровадивши основні принципи контролю версій, як ми розуміємо їх сьогодні [29]. Однак ця система знову ж таки була обмежена одним користувачем і не мала можливостей для підтримки середовищ спільної розробки.

Потреба у спільних інструментах для розробки програмного забезпечення призвела до створення системи контролю ревізій (Revision Control System, RCS) на початку 1980-х років Вальтером Ф. Тіхі. RCS покращила SCCS, запровадивши більш ефективні механізми зберігання та кращу підтримку декількох користувачів [15]. Незважаючи на ці удосконалення, як SCCS, так і RCS були в основному локальними системами контролю версій, тобто працювали на одній машині і не були призначені для мережевих середовищ.

Історичний контекст систем контролю версій визначався безперервними зусиллями адаптації до мінливих потреб процесів розробки програмного забезпечення. Від ручних методів до локальних систем контролю версій, а пізніше до централізованих і розподілених систем, кожен етап є реакцією на конкретні виклики, зумовлені еволюцією обчислювальної техніки та практик розробки програмного забезпечення. Розуміння цієї історії є важливим для осягнення поточного стану та майбутнього потенціалу систем контролю версій у керуванні дедалі складнішими колективними процессами розробки програмного забезпечення і документації.

### Ручні методи контролю версій

Витоки систем контролю версій можна простежити ще до цифрової ери, коли ручні методи та контроль документації були основними засобами управління версіями документів та програмного забезпечення. Розглянемо ці архаїчні, проте фундаментальні методи а надамо історичний контекст еволюції систем контролю версій.

Спочатку контроль версій був переважно ручним процесом. В контексті управління документами та програмним забезпеченням, це включало фізичне супроводження та організацію паперових документів чи магнітних стрічок, де зберігалися коди програмного забезпечення. Кожна ревізія документа чи компонента програмного забезпечення була окремим фізичним об'єктом, часто позначена або анотована від руки, щоб вказати її версію, дату створення та автора. Цей примітивний підхід був трудомістким і призводив до помилок, але заклав основу для більш досконалих систем.

В академічних і корпоративних середовищах, особливо в інженерії та розробці програмного забезпечення і документації, відстеження змін було вкрай важливим. Інженери і розробники вели "журнали змін" або "журнали модифікацій" у своїй документації, практика, яка досі поширена в різних формах [30]. Ці журнали забезпечували хронологічний запис змін, виправлення помилок та додавання нових можливостей, слугуючи примітивною формою відстеження версій.

Контроль документації був ще одним визначальним аспектом раннього контролю версій. Це включало суворі правила управління документами, зокрема створення, перегляд, модифікацію, випуск, розповсюдження та доступність документів. У багатьох галузях, особливо в таких як аерокосмічна, оборонна та галузь охорони здоров'я, контроль документації був і залишається регуляторною вимогою, що забезпечує точність та відстежуваність інформації з плином часу [31].

У цих системах кожна ревізія документа, як правило, копіювалася, переглядалася та затверджувалася перед тим, як бути "випущеною" для використання. Оригінал документа часто зберігався в безпечному місці, в той час як копії розповсюджувалися серед відповідних сторін. Цей метод гарантував, що всі працювали з "останньою" версією, концепція, що є центральною для сучасних систем контролю версій.

Ручні методи та контроль документації були ефективними до певної міри, але мали значні обмеження, особливо зі зростанням складності та розміру проектів. Трудомісткий характер цих методів, разом із ризиком людської помилки та труднощами у відстеженні множинних змін одночасно, призвели до виникнення потреби в більш систематичному контролі версій у розробці програмного забезпечення. Ця потреба ставала все більш виразною з появою цифрової ери та поширенням використання комп'ютерів наприкінці 20-го століття.

З просуванням в еру цифрових обчислень, ці ручні методи та контроль документації заклали основоположні принципи для першого покоління систем контролю версій. Перехід від ручних до автоматизованих систем позначив значний стрибок в ефективності та результативності контролю версій, підготувавши фундамент для більш досконалих систем, які ми використовуємо сьогодні.

### Систематичний конроль

Необхідність систематичного контролю версій у розробці програмного забезпечення і документації виникла як пряма відповідь на зростаючу складність і колаборативний характер проектів. До цього розробка програмного забезпечення і документації часто була більш лінійною та індивідуалістичною справою, де виклики управління різними версіями коду були менш помітними. Однак, по мірі зростання складності і масштабів систем програмного забезпечення, розширення команд та створення географічно розподілених команд, обмеження неформальних методів контролю версій ставали дедалі очевиднішими.

Із зростанням розміру і складності проектів програмного забезпечення і документації, завдання відстеження змін, управління множинними версіями коду та забезпечення узгодженості між різними частинами проекту ставало дедалі складнішим. Ранні методи контролю версій, часто орієнтовані на ручні документо-орієнтовані не автоматизовані процеси, були недостатніми для зростаючих вимог. Необхідність більш систематичного підходу була підкреслена декількома сумнозвісними збоями програмного забезпечення і помилками, які були наслідком поганого управління версіями та відсутності координації між розробниками [29].

Перехід до колаборативних середовищ розробки, особливо з появою проектів з відкритим кодом і розподілених команд, ще більше підкреслив необхідність систематичного контролю версій. Саме тоді здатність ефективно співпрацювати над кодовими базами, не перезаписуючи роботу один одного та не гублячи зміни, стала критичною вимогою [30]. Це вимагало системи, яка не лише відстежує зміни, але й полегшує співпрацю між декількома розробниками, можливо розташованими в різних географічних локаціях.

Прийняття ітеративних та інкрементних моделей розробки, таких як гнучкі методології (Agile), також відіграло важливу роль у виникненні систематичного контролю версій. Ці моделі розробки, які підкреслюють адаптивність і часті зміни програмного забезпечення і документації, вимагали більш динамічного і гнучкого підходу до контролю версій. Традиційні методи були занадто повільними і неефективними, щоб пристосуватись до швидкого темпу змін, притаманних цим методологіям [32].

Технологічні досягнення, особливо в сфері обчислювальних потужностей та зберігання даних, також сприяли можливості і необхідності більш досконалих систем контролю версій. Збільшені можливості зберігання та обробки даних дозволили розробку більш складних систем контролю версій, здатних опрацьовувати великі кодові бази та детальну історію змін [33].

Отже, поява необхідності систематичного контролю версій у розробці програмного забезпечення була багатогранним розвитком, спричиненим зростаючою складністю проектів програмного забезпечення, появою колаборативних і розподілених команд розробки, прийняттям ітеративних моделей розробки та технологічним прогресом. Ця потреба призвела до розробки першого покоління систем контролю версій, заклавши фундамент для еволюції більш просунутих систем у наступні роки.

### Перше покоління: локальні системи контролю версій

Перше покоління цих систем, відомих як локальні системи контролю версій, ознаменувало значний прогрес у галузі програмної інженерії. Ці системи були створені для вирішення проблем, які ставили попередні, ручні методи контролю, що часто були громіздкими і призводили до помилок [30].

Локальні системи контролю версій працювали в межах персонального комп'ютера користувача. На відміну від ручних методів, які покладалися на копіювання файлів у окремі каталоги для управління версіями, л автоматизували цей процес. Одним з найперших прикладів локальної системи контролю версій була система контролю вихідного коду (Source Code Control System, SCCS), розроблена Марком Дж. Рочкіндом у Bell Labs в 1972 році. SCCS ввела концепцію зберігання лише змін, зроблених у програмному забезпеченні, а не повних копій файлів, метод, відомий як «зберігання різниці» або «диференційне зберігання» [29]. Цей підхід значно зменшив об'єм сховища, необхідного для зберігання кожної версії.

Іншою помітною системою цієї ери була Система Контролю Ревізій (Revision Control System, RCS), яка розвивалася на основі SCCS. Розроблена Вальтером Ф. Тічі на початку 1980-х років, RCS вдосконалила SCCS, впровадивши більш ефективний алгоритм зберігання та кращі інструменти для злиття змін та вирішення конфліктів [15]. RCS мала значний вплив на архітектуру наступних систем контролю версій, оскільки заклала основи для багатьох концепцій, які досі використовуються в сучасних системах, таких як ідентифікатори комітів та атомарні операції.

Основною перевагою цих ранніх систем була їх простота та контроль, який вони надавали окремим розробникам над версіями їх коду. Однак у них також були значні обмеження. Найбільш помітним була їх відсутність підтримки спільної роботи. Оскільки ці системи були локальними, контроль версій кожного розробника був ізольований, що ускладнювало інтеграцію змін з декількох джерел. Це обмеження ставало все більш проблематичним, оскільки програмні проекти зростали за розміром та складністю, вимагаючи співпраці між більшими командами розробників.

Таким чином, локальні системи контролю версій стали важливим кроком в еволюції технологій контролю версій. Вони перевели контроль версій з ручного процесу до більш надійної автоматизованих систем. Хоча вони мали обмежені можливості підтримки спільної розробки, ці системи ввели ключові концепції та методології, які проклали шлях для більш просунутих, колаборативних систем контролю версій в наступних поколіннях.

### Друге покоління: централізовані системи контролю версій

Еволюція систем контролю версій ознаменувалася важливою віхою з появою другого покоління, яке характеризується централізованими системами контролю версій (ЦСКВ). Цей етап вирішував численні обмеження, властиві системам першого покоління.

Централізовані системи контролю версій з'явилися у відповідь на зростаючу потребу у співпраці в проектах розробки програмного забезпечення. У централізованих системах головна частина встановлюється на сервері, де зберігається вся інформація, а розробники як клієнти підключаються до цього сервера, щоб отримати останню версію коду та внести зміни [30]. Така архітектура стала значним вдосконаленням в порівнянні з локальними локальними системами контролю версії, оскільки дозволяла декільком розробникам працювати над одним проектом, зменшуючи ризики конфліктних змін.

Одним з найбільш помітних прикладів централізованої системи контролю верссій є Subversion (SVN), яка була спеціально розроблена як вдосконалена CVS (Concurrent Versions System) - перша широко поширена централізована система контролю версій [30]. SVN пропонувала декілька вдосконалень в порівнянні з CVS, таких як атомарні коміти (що гарантують, що або всі зміни в коміті застосовуються, або жодна не застосовується), та кращу роботу з бінарними файлами. Централізована природа SVN та подібних систем також означала, що адміністратори могли реалізувати більш тонкий контроль над дозволами доступу - важлива функція для великих команд [34].

Проте, незважаючи на свої переваги, централізовані системи також мали обмеження. Найбільш значущим була одна точка відмови; якщо центральний сервер виходив з ладу, ніхто не міг співпрацювати або отримати доступ до останньої версії проекту. Більше того, ці системи часто вимагали постійного підключення до сервера, що могло бути перешкодою для розробників, які працюють віддалено або в регіонах з нестабільним інтернет-з’єднанням [19].

В академічній літературі вивчення централізованих систем контролю версій часто зосереджується на їх ролі у сприянні спільній роботі, а також висвітлює проблеми, які вони створюють у плані залежності від сервера та потенційних "вузьких місць" у робочому процесі [33]. Незважаючи на ці проблеми, централізована модель була важливим кроком в еволюції контролю версій, підготувавши фундамент для наступного покоління систем контролю версій - розподілених систем.

Отже, друге покоління систем контролю версій, представлене централізованими моделями на зразок SVN, відіграло вирішальну роль в удосконаленні процесів спільної розробки програмного забезпечення. Хоча вони вирішили багато обмежень локальних систем, їх залежність від центрального сервера створила нові проблеми, відкривши шлях для розвитку розподілених систем контролю версій.

### Третє покоління: розподілені системи контролю версій

Наступним етапом еволюції систем контролю версій стала поява третього покоління, яке характеризується розподіленими системами контролю версій. Це покоління позначило зміну парадигми в управлінні контролем версій, пропонуючи розширені можливості та вирішуючи багато обмежень, властивих централізованій моделі другого покоління.

Централізовані системи, хоча й були покращенням у порівнянні з локальними системами контролю версій, все ще мали єдину точку відмови у центральному сховищі та часто отримали нарікання за їх недостатню продуктивність, особливо у масштабних проектах [35]. Розподілені системи контролю версій з'явилися як відповідь на ці проблеми, пропонуючи більш надійний, гнучкий та ефективний підхід до контролю версій.

На відміну від своїх централізованих аналогів, розподілені системи контролю версій, такі як Git, Mercurial та Bazaar, не зберігають всю інформацію на одному сервері. Натомість кожен учасник має повну локальну копію всіх версій даних, включаючи всю історію змін [19]. Ця децентралізація фундаментально змінює робочий процес кількома способами:

* **Надлишковість та стійкість**: розподілена природа властиво забезпечує надлишковість. Кожна копія сховища є повним резервним копіюванням, підвищуючи стійкість до втрати даних [19].
* **Продуктивність та гнучкість**: операції, такі як коміт, галуження (створення гілок) та злиття, виконуються локально, що призводить до значного підвищення продуктивності. Це також дозволяє розробникам працювати автономно, синхронізуючи зміни інших у зручний час [35].
* **Спрощена модель створення гілок**: розподілені системи контролю версій полегшує більш складні практики галуження та злиття, дозволяючи більш динамічну модель співпраці. Це призвело до розробки різних робочих процесів, таких як гілки функцій та Gitflow, які особливо підходять для розподілених систем контролю версій [36].

Поширення розподілених систем контролю версій мало глибокий вплив на практики розробки програмного забезпечення. Удосконалений, та більш гнучкий підхід до контролю версій, краще пристосований для розподілених команд та гнучких методологій, ніж попередні покоління, значно спрощував процес паралельної розробки в розподілених командах, а також за рахунок простоти процесу створення гілок заохочував до експериментів у розробців, сприяючи інноваціям.

Незважаючи на свої переваги, розподілені системи також не позбавлені проблем. Крива навчання, особливо для таких систем, як Git, може бути досить крутою для користувачів, звиклих до централізованих систем. Крім того, децентралізований підхід може призвести до безконтрольного розмноження гілок і, без належного управління, до більш складної історії змін.

Таким чином, третє покоління систем контролю версій, революціонізувало сферу розробки програмного забезпечення і документації. Долаючи багато обмежень попередніх поколінь та впроваджуючи нові можливості з точки зору продуктивності, гнучкості паралельної розробки та співпраці в розподілених командах, розподілені системи контролю версій стають фактичним стандартом для сучасних проектів розробки програмного забезпечення і документації. Оскільки галузь продовжує розвиватися, робимо висновок, що принципи та технології, що лежать в основі розподілених систем контролю версій, будуть мати значний вплив і на майбутній розвиток систем контролю версій.

### Сучасні тенденції та перспективи на майбутнє

Ландшафт систем контролю версій постійно розвивається, формуючись під впливом динамічних потреб розробки програмного забезпечення та технологічних досягнень у сфері обчислювальної техніки. Зараз можна виділити кілька тенденцій та потенційних майбутніх розробок.

**Сучасні тенденції**:

* **Інтеграція з DevOps та процесами безперервної інтеграції / безперервним розгортанням (CI/CD)**. Сучасні системи контролю версій все глибше і щільніше інтегруються з практиками та інструментами DevOps. Така інтеграція сприяє безперервній інтеграції та безперервному розгортанню, дозволяючи більш ефективні та автоматизовані робочі процеси [37]. Інструменти як Jenkins, Travis CI та GitLab CI наочно демонструють тенденцію, де контроль версій глибоко інтегрується з усім життєвим циклом розробки програмного забезпечення.
* **Застосування машинного навчання та штучного інтелекту**. Спостерігається зростаючий інтерес до застосування машинного навчання (ML) та штучного інтелекту (AI) до різних аспектів контролю версій, таких як автоматизація перегляду коду, прогнозування потенційних конфліктів та оптимізація управління гілками [38]. Ці технології можуть значно підвищити ефективність та результативність систем контролю версій.
* **Посилення уваги до безпеки та дотримання регуляторних вимог**. Зі зростанням кіберзагроз та регуляторних вимог, сучасні системи контролю версій все частіше вбудовують передові функції безпеки. Основні з яких - це удосконалений контроль доступу, шифрування та інтеграцію з інструментами тестування безпеки, забезпечення захисту кодової бази та дотримання усіх формальних регуляторних вимог [39].
* **Покращення користувацького досвіду та інструментів співпраці**. Спостерігається спроба зробити системи контролю версій більш зручними не тільки для програмістів, але й для інших користувачів орієнтованих на співпрацю. Платформи як GitHub та GitLab безперервно вдосконалюють свої користувацькі інтерфейси та пропонують додаткові функції управління проектами та співпраці в режимі реального часу, полегшуючи командну роботу [40].

**Перспективи на майбутнє:**

* **Децентралізація за межами управління кодом**. Концепція децентралізації, визначальна риса розподілених систем, таких як Git, може поширитися за межі управління кодом. Майбутні системи можуть децентралізувати інші аспекти розробки програмного забезпечення, такі як документація та управління проектами, пропонуючи більш інтегроване та спільне середовище.
* **Хмаро-орієнтовані системи контролю версій**. В міру того, як хмарні обчислення продовжують домінувати, ми можемо побачити появу хмаро-орієнтованих систем контролю версій, спеціально розроблених для використання переваг хмарних, кластеризованих та мікросервісних архітектур: масштабованості, продуктивності та надійності [41].
* **Крос-платформне та мультирепозиторне управління**. Зі зростаючою складністю програмних проектів, майбутні системи контролю версій мають запропонувати більш надійні рішення для управління декількома репозиторіями на різних платформах, спрощуючи процес розробки великомасштабних мультиплатформених проектів.
* **Генерація та управління кодом за допомогою штучного інтелекту**. Інтеграція штучного інтелекту в системи контролю версій може еволюціонувати не лише для допомоги в управлінні кодом, а й для генерування та оптимізації коду, що потенційно може революціонізувати сам процес розробки програмного забезпечення.
* **Квантові обчислення**. Із розвитком квантових обчислень, системи контролю версій потребуватимуть адаптації для управління та відстеження квантових алгоритмів та програмного забезпечення, які можуть мати фундаментально інші вимоги та виклики у порівнянні з класичним програмним забезпеченням.

Підсумовуючи, майбутнє систем контролю версій, схоже, рухається в напрямку більш інтегрованих, інтелектуальних та зручних для користувача рішень, які стають невідʼємними частинами процесу розробки програмного забезпечення, документацій та управління проектами. Безперервна еволюція цих систем імовірно відіграватиме вирішальну роль у формуванні того, як буде розроблятись, розгортатись та підтримуватись програмне забезпечення в наступні роки.

## Основні терміни

У сфері систем контролю версій розуміння специфічної термінології є вкрай важливим для засвоєння основних принципів і операцій. Цей розділ пояснює ключові терміни, що є фундаментальними для дискурсу щодо систем контролю версій.

* **Репозиторій**: Репозиторій, часто скорочено «репо», є сховищем файлів, де зберігаються файли в системі контролю версій. Його можна розглядати як базу даних змін, що дозволяє користувачам отримувати попередні версії файлів і відстежувати модифікації з часом [35].
* **Різниця або зміна (diff**): Скорочено від «difference», процес порівняння двох версій файлу із спеціальним позначенням для виділення зміни даних від однієї версії до іншої, а також сам результат такого порівняння, тобто різниця між двома версіями. Ця функція є надзвичайно важливою для розуміння конкретних змін, зроблених в кожній ітерації файлу [33].

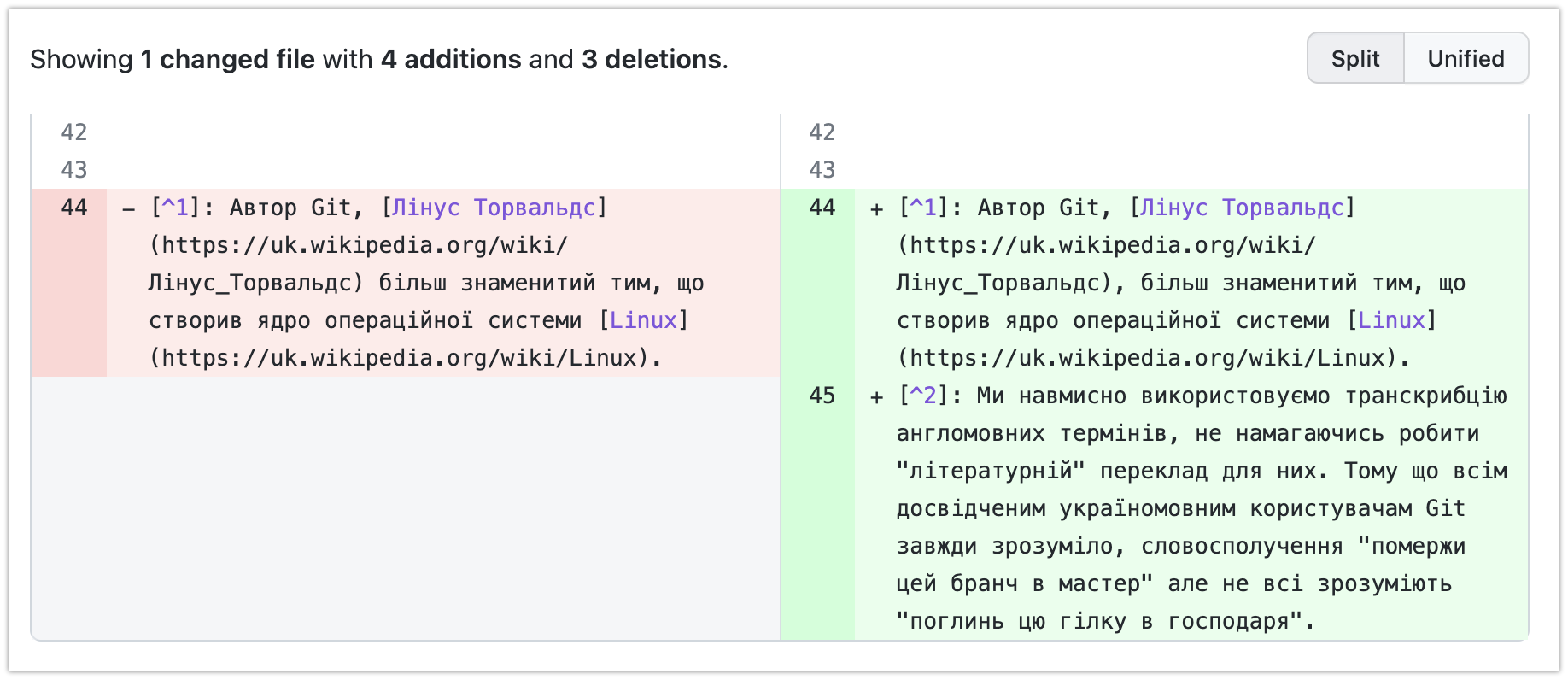


Рисунок 3.1 – Відображення різниці між версіями файл у GitHub [45]

* **Патч (patch)**: Патч є файлом, що складається зі списку змін (diffs) і використовується для трансформації однієї версії файлу (або набору файлів) в іншу. Патчі часто використовуються для пропонування змін у спільних проектах або для оновлення програмного забезпечення [42].
* **Коміт (commit)**: У системах контролю версій коміт є дією, що фіксує зміни в репозиторії. Кожен коміт має унікальний в межах репозиторію ідентифікатор (часто хеш-суму від зафіксованих даних), який дозволяє відрізняти одну версію від іншої і відстежувати зміни між ними. Це схоже на знімок вашого проекту в певний момент [19].

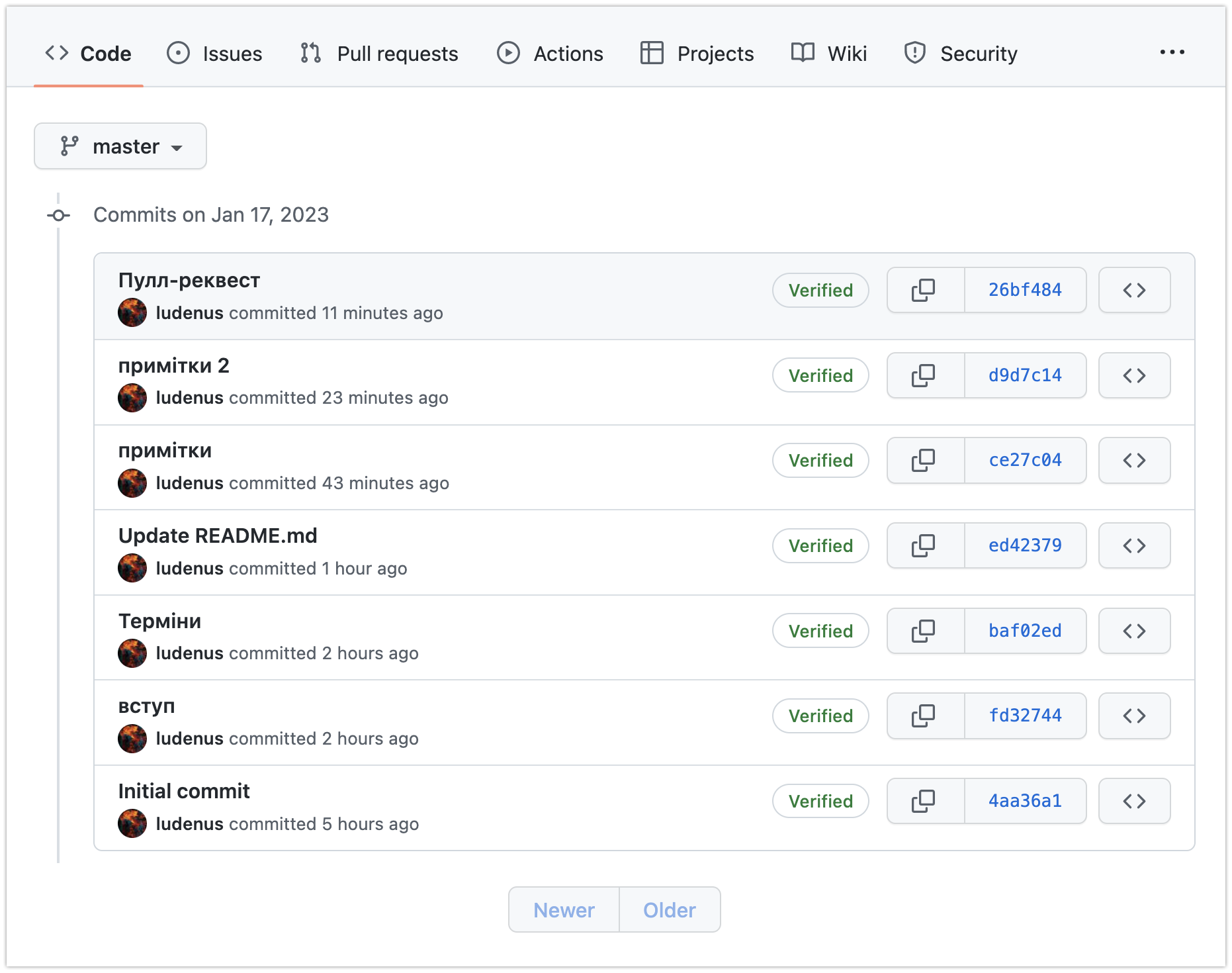


Рисунок 3.2 – Список комітів в GitHub [45]

* **Тег (tag)**: Тег є маркером, що позначає певну важливу точку в історії репозиторію, наприклад випуск стабільної версії. Теги часто використовуються для версіювання, дозволяючи легший доступ до певних версій програмного коду або документації [30].

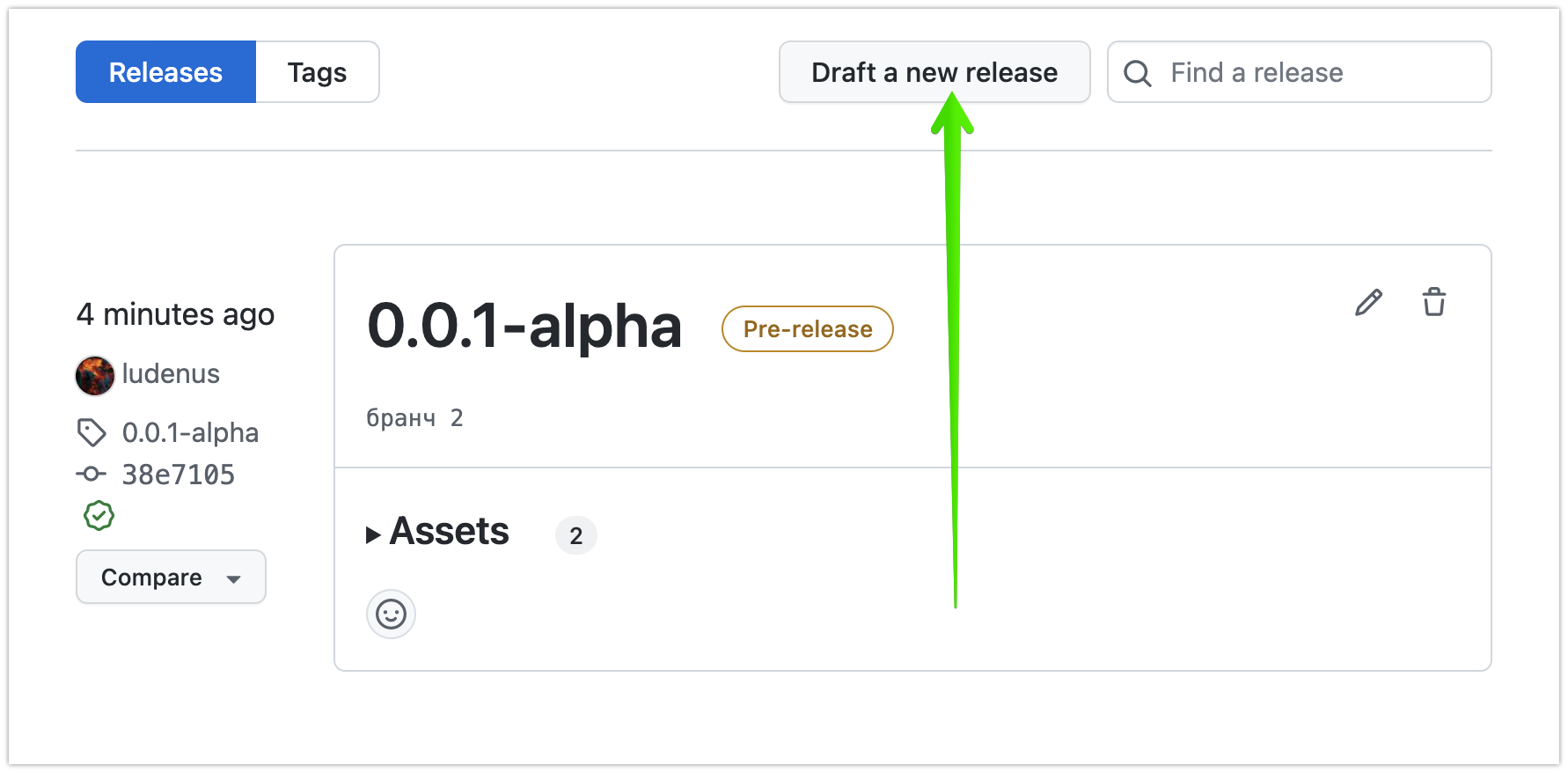
[](https://github.com/ludenus/uad-multimedia-doc/blob/master/img/uad-multimedia-doc-0050-release.png)

Рисунок 3.3 – Створення нового тегу в GitHub [45]

* **Гілка (branch)**: Відгалуження, яке передбачає відхід від основної лінії розробки і продовження роботи в автономному режимі, без внесення змін в основну лінію. Може відрізнятися від основної лінії змінами в файлах, іншим набором комітів, або іншою їх послідовністю. Після завершення автономної роботи (і перевірки роботоспроможності версії), зміни можуть бути злиті в основну гілку розробки. Ця концепція є вкрай важливою для підтримки паралельної розробки серед декількох членів команди [43].

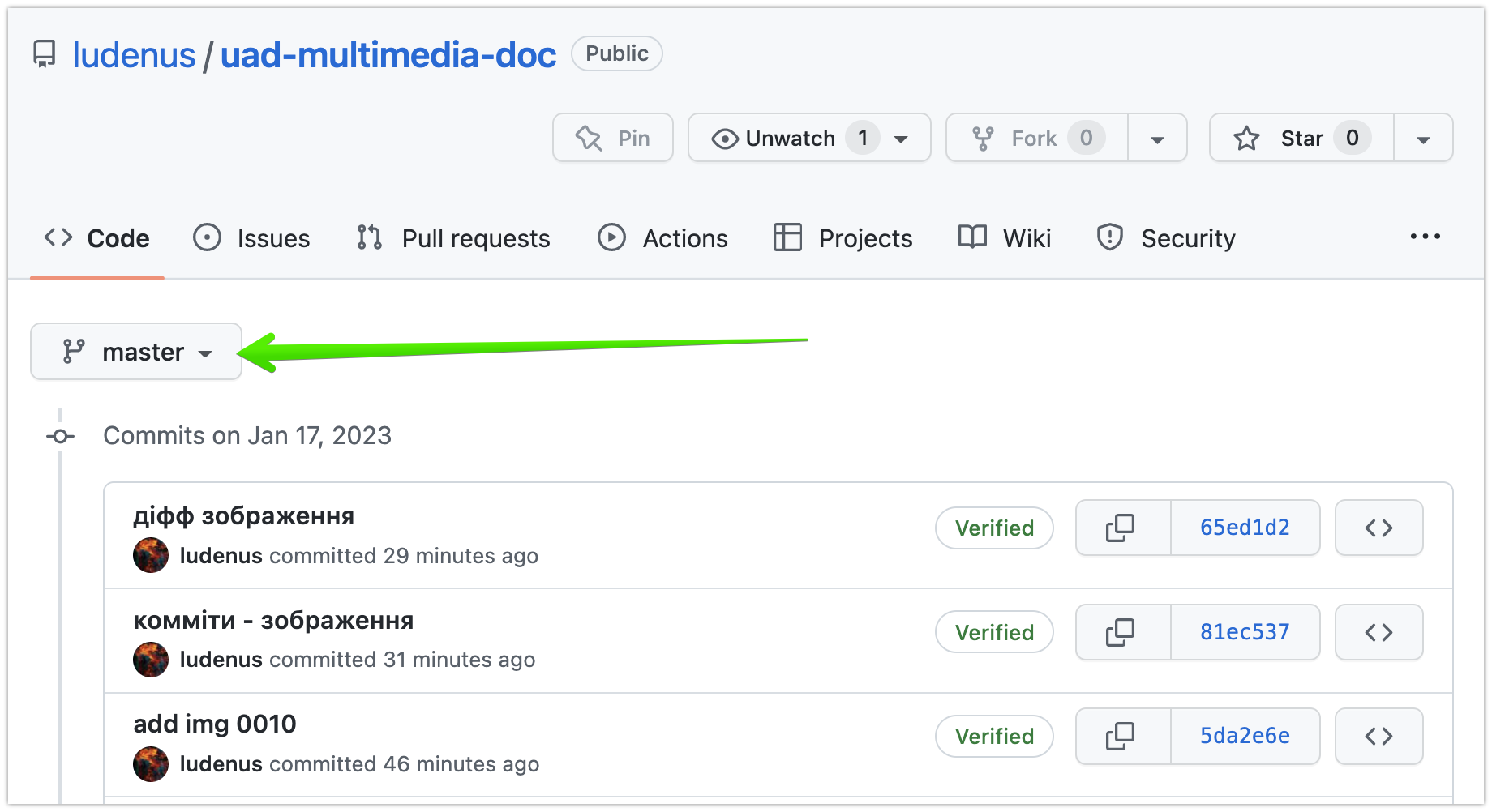


Рисунок 3.4 – Основна гілка розробки в GitHub [45]

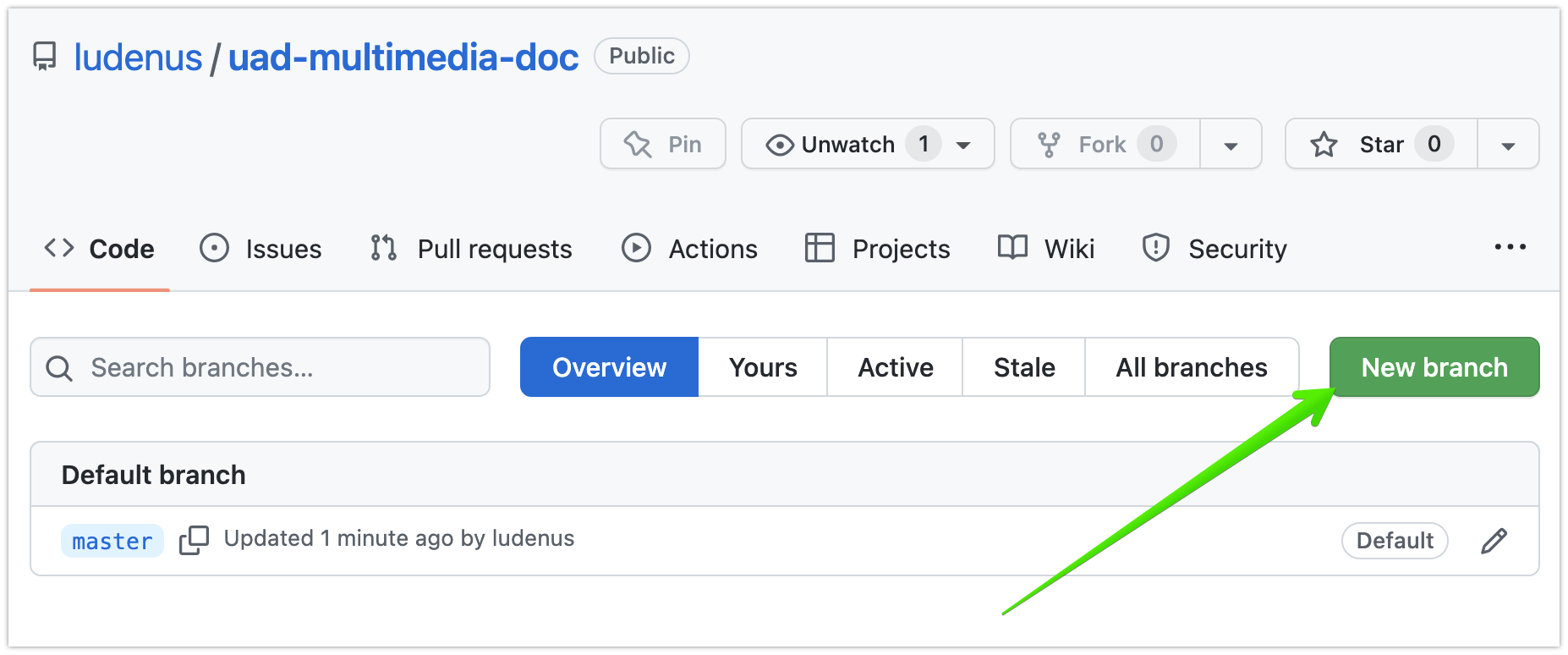


Рисунок 3.5 – Створення нової гілки в GitHub [45]

* **Злиття (merge)**: Злиття - це процес інтеграції змін з різних гілок назад в єдину гілку, часто основну. Цей процес іноді може призводити до конфліктів, які потребують вирішення [19].
* **Запит на злиття (pull request, merge request)**: Запит на злиття (або запит на об'єднання в деяких системах) є способом внесення змін до основної гілки проекту. Він сповіщає членів команди, що окремий користувач завершив свій набір змін і просить об'єднати ці зміни з основною гілкою [40].

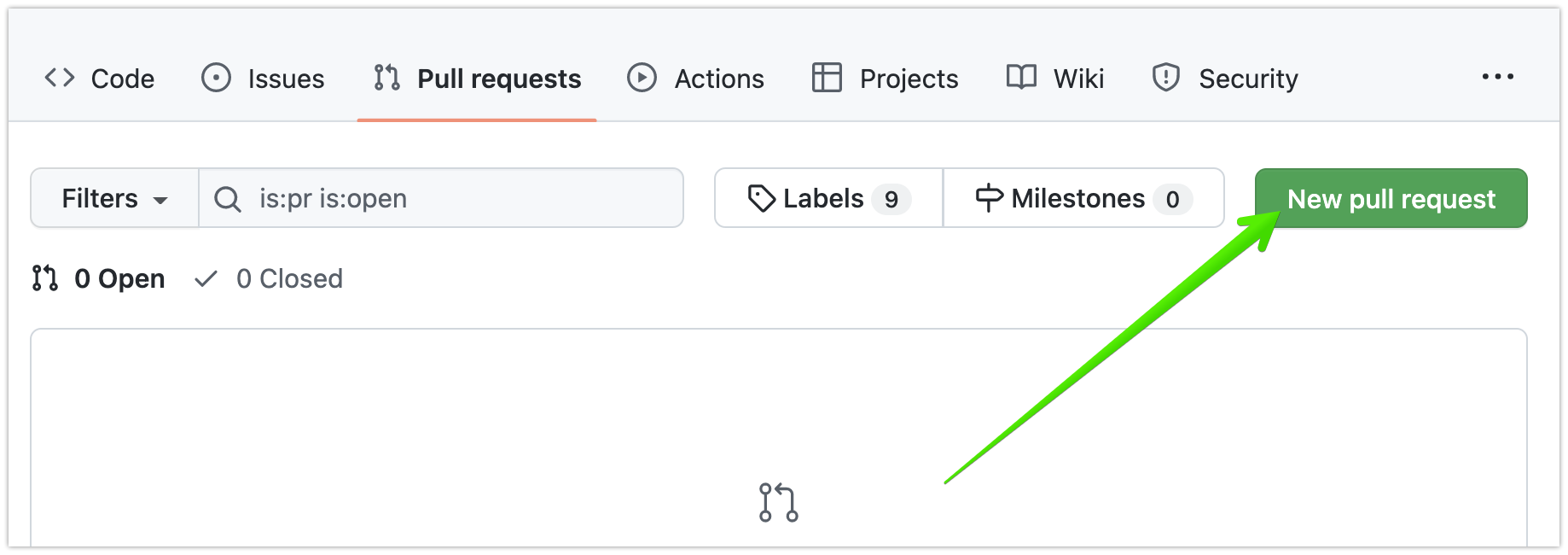
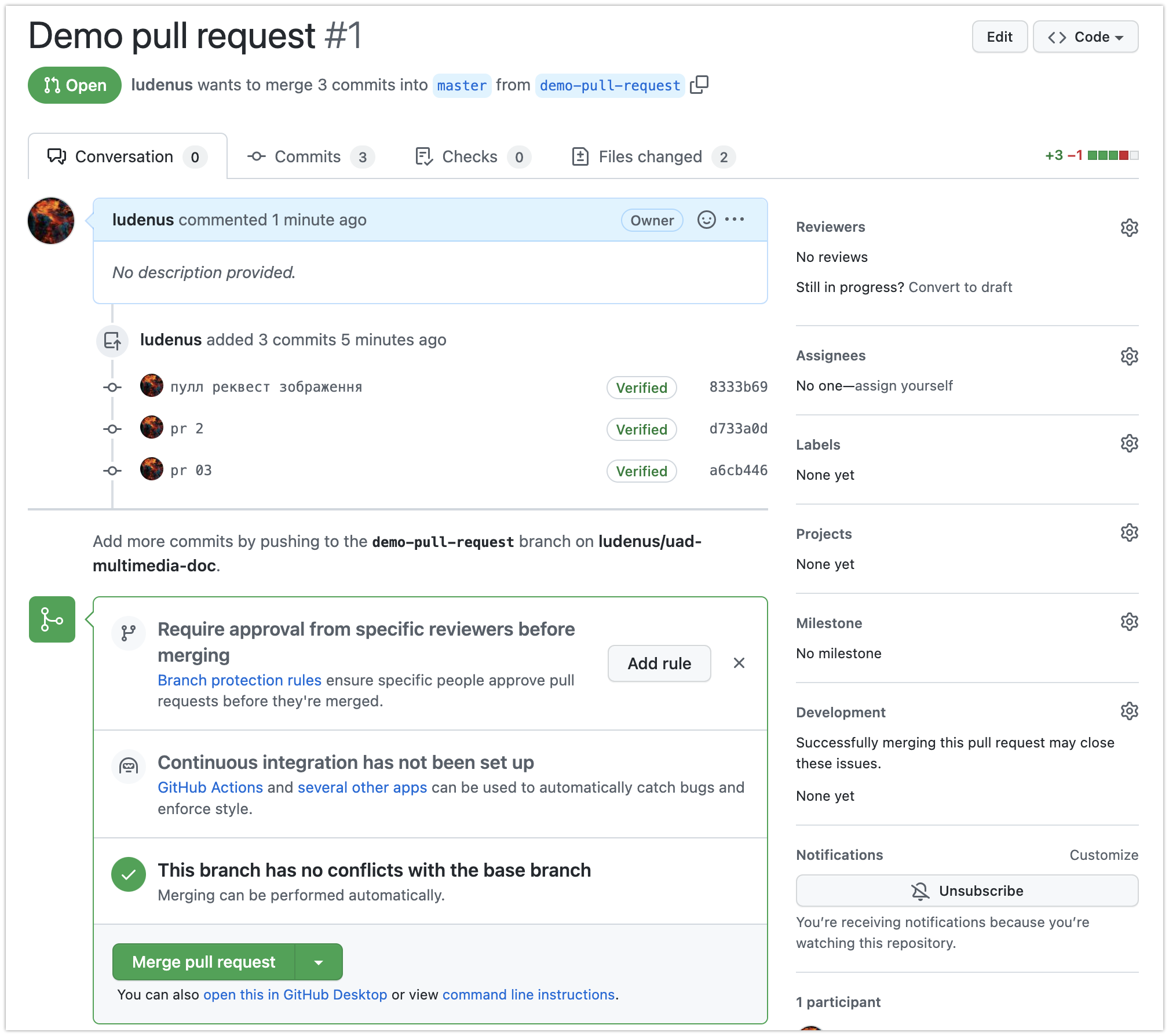


Рисунок 3.6 – Створення запиту на злиття в GitHub [45]

Зазвичай запит на злиття застосовується, коли над проектом працює декілька людей. Типово заохочує інших учасників проекту робити перегляд змін і надавати зворотній зв'язок автору. Запит на злиття може бути прийнято (бранч користувача буде злитий з основною гілкою), відхилено (пропоновані зміни однозначно небажані з точки зору інших співавторів), або відмічено для подальшого обговорення і вдосконалення запропонованих змін.

  
Рисунок 3.7 – Перегляд запиту на злиття в GitHub [45]

* **Конфлікт зливання (merge conflict)**: В контексті систем контролю версій конфлікт виникає, коли зміни в одній гілці несумісні зі змінами в іншій, таким чином вони роблять неможливим злиття. Вирішення конфліктів вимагає ручного втручання в процес злиття, зазвичай потребує участі декількох співавторів, що спільно працюють над кодом і є критичним аспектом спільної роботи в розподілених системах контролю версій [44].

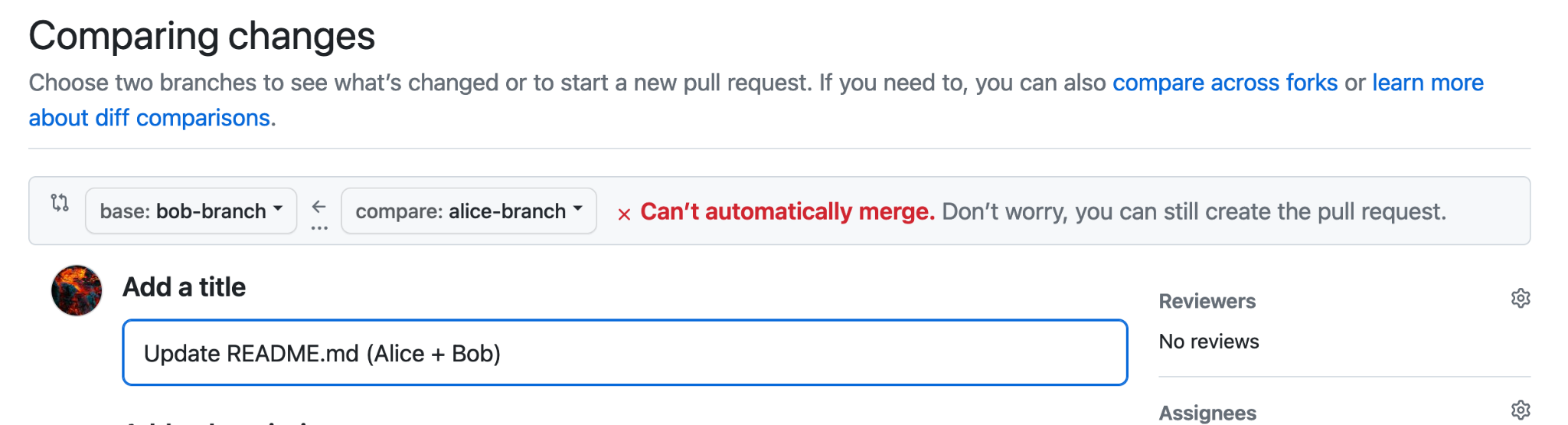


Рисунок 3.9 – Конфлікт, виявлений при порівнянні гілок в GitHub [45]

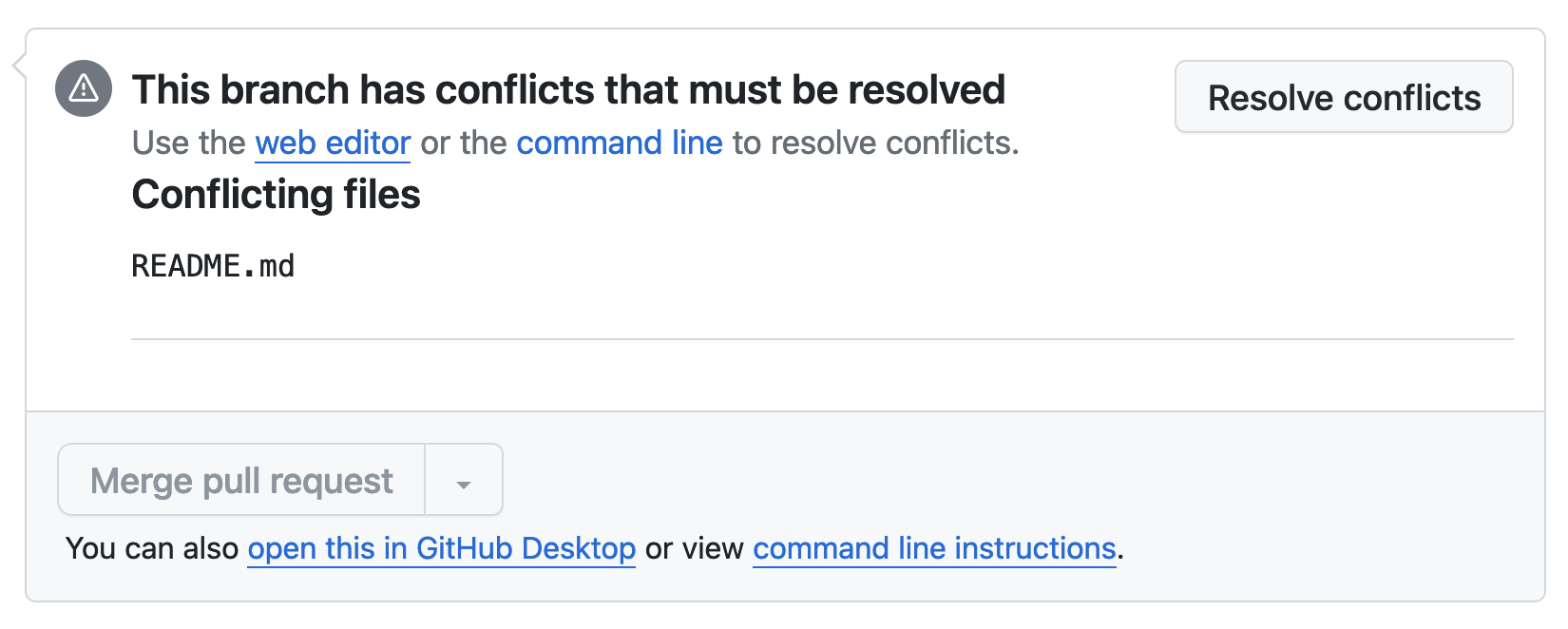
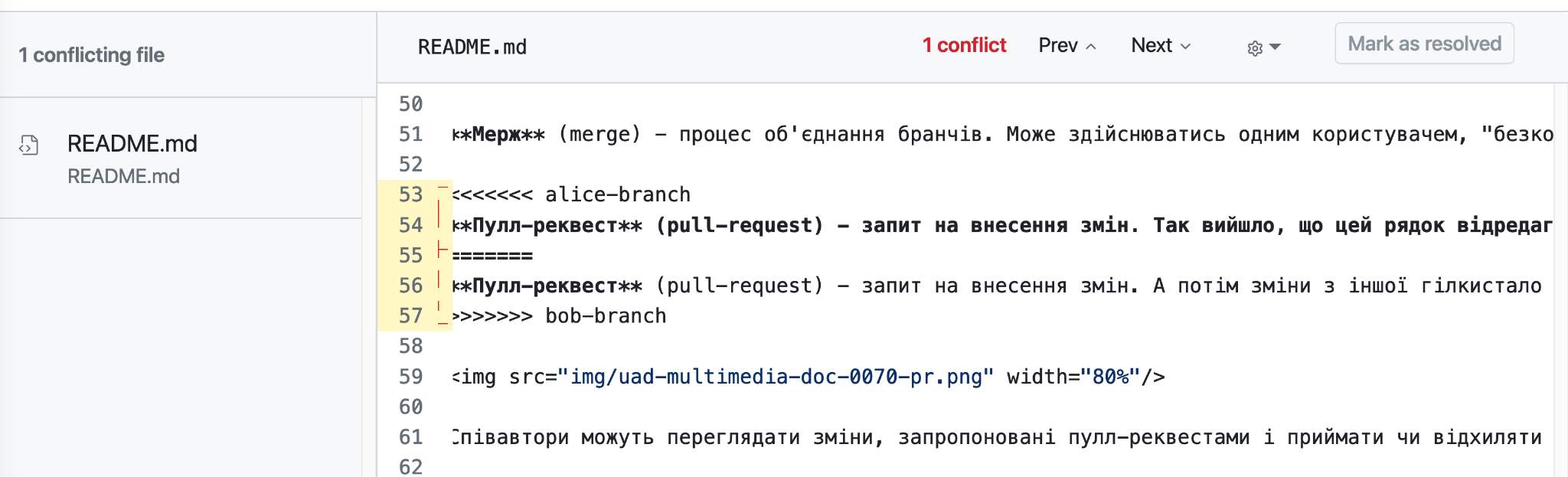


Рисунок 3.9 – Файли, зміни в яких спричиняють конфлікт в GitHub [45]

  
Рисунок 3.10 – Ручне редагування для вирішення конфлікту злиття в GitHub [45]

Розуміння цих термінів забезпечує базовий лексикон для поглиблення в принципи та функціональні можливості систем контролю версій. Кожен термін представляє критичний компонент у складному процесі управління та відстеження змін у розробці програмного забезпечення та документації.

## Основні поняття

* **Контроль версій (Revision Control, Version Control)**. Відомий також як управління версіями або контроль вихідного коду, це управління змінами в документах, комп’ютерних програмах, великих веб-сайтах та інших наборах інформації. Це фундаментальний компонент систем контролю версій. Кожну модифікацію як правило називають «ревізією» або «версією». Кожна версія асоціюється з міткою часу та особою, яка внесла зміну. Версії можуть порівнюватися, відновлюватися, а з деякими типами файлів об'єднуватися [17]. Саме контроль версій надає можливість повернутися до попередньої ревізії документа, що є вирішальним для відстеження правок інших редакторів, виправлення помилок та захисту від вандалізму і спаму в спільних середовищах як Вікіпедія [46].
* **Управління вихідним кодом (Source Code Management, SCM)**. Це підмножина контролю версій, зосереджена конкретно на управлінні змінами під час розробки програмного забезпечення. Практики управління вихідним кодом включають контроль версій та встановлення базових ліній. Якщо щось піде не так, SCM може визначити, що було змінено і хто змінив. Якщо конфігурація працює добре, SCM може визначити як її відтворити на багатьох хостах. Саме поява таких SCM інструментів як Git та Subversion значно підвищила ефективність і швидкість розробки програмного забезпечення [35].
* **Відстеження змін (Change Tracking)**. Передбачає ведення записів змін, зроблених до документа чи набору файлів протягом всього часу існування проекту. Цей процес є надзвичайно важливим у розробці програмного забезпечення і документації, де необхідно відстежувати кожну зміну, ретельно фіксувати, коли і хто її зробив і чому вона була зроблена. Цей історичний запис може допомогти вирішити питання чи проблеми, які виникають пізніше в процесі розробки, а також є вкрай необхідним для аудиторських цілей [19].
* **Розгалуження та злиття (Branching and Merging)**. Розгалуження - це дублювання об’єкта в системі контролю версій (такого як файл вихідного коду чи дерево каталогів), щоб модифікації могли відбуватися паралельно в декількох незалежних гілках. Злиття - це процес поєднання двох або більше наборів даних в єдиний уніфікований набір. В контексті розробки програмного забезпечення, розгалуження і злиття дозволяють розробникам працювати незалежно над функціями чи виправленнями, а потім об'єднувати їхні зміни назад до основного коду, процес, який став більш плавним з появою розподілених систем контролю версій, таких як Git [47].
* **Вирішення конфліктів (Conflict Resolution)** Відбувається, коли зміни від різних учасників не співпадають і не можуть бути автоматично об'єднані. Така ситуація вимагає ручного втручання для вирішення розбіжностей. Ефективні стратегії вирішення конфліктів є вкрай важливими для збереження цілісності кодової бази та гарантії, що програмне забезпечення залишається функціональним, а документація консистентною після об'єднання змін із різних джерел. Такі інструменти систем контролю версій як індикатори конфліктів злиття допомагають у виявленні та вирішенні цих конфліктів [44].

Отже, ці основні поняття складають фундамент систем контролю версій, забезпечуючи системний структурований підхід до управління змінами в розробці програмного забезпечення, документації та інших спільних проектах. Вони дозволяють командам працювати ефективніше, точно відстежувати зміни та підтримувати якість та високий рівень контролю над еволюцією своїх проектів.

## Основні принципи роботи систем контролю версій

**Централізовані та розподілені системи.** Операційні принципи сучасних систем контролю версій можна фундаментально класифікувати на два типи: централізовані та розподілені. Централізовані системи контролю версій, такі як Subversion (SVN), працюють за моделлю, де всі дані та історія версій зберігається на центральному сервері. При цьому кожен користувач для роботи копіює на свою локальну машину певну версію з цього центрального сховища [30]. Ця модель спрощує адміністрування, але створює єдину точку відмови та робить неможливою роботу, коли центральний сервер недоступний. Розподілені системи контролю версій, такі як Git та Mercurial, дозволяють кожному користувачеві мати повну копію усього сховища, включаючи повну історію всіх версій, тим самим усуваючи залежність від центрального сервера та за рахунок такої надлишковості, підвищувати надійність збереження даних [35]. Така децентралізація сприяє більш гнучким робочим процесам в розподілених командах, але вимагає ретельного дотримання процедур галуження та злиття для об'єднання змін від різних авторів, або процедур вирішення конфліктів у випадку ускладнень.

**Атомарні операції.** Атомарні операції в системах контролю версій посилаються на групу операцій, які розглядаються як єдиний блок роботи. Це означає, що або всі операції в групі успішно завершуються, або жодна не виконується, гарантуючи узгодженість стану всіх даних сховища. Наприклад, коли коміт здійснюється в Git, то або всі зміни у цьому коміі застосовуються, або не застосовується жодна з них, запобігаючи таким чином від утворення неконсистентних станів репозиторію [35].

**Контроль версій та теги**. Контроль версій – це основний принцип, що дозволяє користувачам відстежувати зміни з часом, повертатися до попередніх станів та розуміти еволюцію проекту. Кожна зміна, внесена до сховища, як правило, має унікальний ідентифікатор або номер версії. У розподілених системах, таких як Git, це SHA-1 хеш, тоді як у централізованих системах, таких як SVN, це послідовний номер [30].

Теги - функція в системах конролю версій, яка використовується для позначення певних точок в історії сховища як значущих, як правило, для стабільних версій (наприклад, v1.0.0, v2.0.0). Ця практика є важливою для підтримки чіткої історії проекту та для посилання на особливо значні зміни версії програмного забезпечення [19].

**Контроль доступу та безпека**. Контроль доступу є важливим для забезпечення того, щоб лише авторизовані особи могли вносити зміни до сховища. У централізованих системах це, як правило, реалізується сервером, який відповідно з конфігурацією, створеною адміністраторами надає права читання або запису до сховища або його частин. Розподілені системи контролю версій через їх природу часто використовують іншу модель, де контроль доступу на запис контролюється, але доступ на читання, як правило, відкритий [35].

Безпека в системах контролю версій означає не тільки контроль доступу, а й цілісність коду. Це включає в себе забезпечення незмінності історії змін та аутентифікації змін. Наприклад, Git використовує криптографічні методи, такі як SHA та GPG для захисту історії змін [35].

Підсумовуючи, операційні принципи систем контролю версій, що охоплюють централізовані та розподілені системи, атомарні операції, контроль версій та тегів, а також контроль доступу та безпеку, формують основу ефективного контролю версій. Ці принципи гарантують, що системи контролю версій не лише ефективно відстежують та керують змінами, але й підтримують цілісність та безпеку проектів розробки програмного забезпечення і документації.

## Робочі процеси в системах контролю версій

Робочі процеси в системах контролю версій - це структуровані підходи і методи, що допомагають командам в управлінні змінами коду протягом всього часу існування проекту. Вибір варіанту робочого процесу є важливими для координації зусиль команди, особливо у великих проектах, та забезпечення системного підходу до розробки. Цей розділ розглядає основні популярні робочі процеси: робочий процес гілок функцій, робочий процес форків, Git flow та GitHub flow.

**Робочий процес гілок для нових функцій (Feature Branch Workflow)**. Робочий процес гілок функцій є одним із основних у сучасних практиках керування версіями, зокрема в розподілених системах керування версіями, таких як Git. У цьому робочому процесі розробники створюють нову гілку для кожної нової функції чи виправлення помилки, над якими працюють. Такий підхід підтримує стабільність основної кодової бази, і як тільки нова функція в окремій гілці буде завершена і протестована, її зливають назад до основної гілки [35]. Цей робочий процес сприяє прозорому, організованому та контрольованому процесу управління змінами в репозиторії, дозволяючи при цьому паралельну розробку одночасно декількох нових функцій.

**Робочий процес форків репозиторію (Forking Workflow)**. Робочий процес форків - це розподілений робочий процес, що надає кожному розробнику (або групі розробників) свій репозиторій, пропонуючи високий рівень ізоляції для кожного учасника [36]. Цей робочий процес особливо популярний у проектах з відкритим кодом. Учасники копіюють оригінальний репозиторій повністю, вносять необхідні зміни у свій репозиторій, а потім пропонують ці зміни до оригінального репозиторію через запит на злиття (pull request). Ця модель є вкрай ефективною в проектах з великою кількістю учасників, оскільки дозволяє вносити масштабні, значущі зміни не лише від окремих розробників і співавторів, а і від цілих команд і окремих компаній.

**Git Flow***.* Git Flow - це модель галуження, спеціально розроблена для Git. Вона визначає чітку модель галуження, яку було спроектовано навколо процесу релізу проекту. Git Flow забезпечує надійну структуру для управління більш масштабними проектами, особливо тими, що мають чітко заплановані цикли випуску нових версій [36]. Основні принципи Git Flow включають:

1. **Головна гілка (Master Branch):** Ця гілка містить код, який завжди знаходиться в стані готовності до випуску. Зазвичай вона використовується для випуску стабільних версій продукту.
2. **Гілка розробки (Develop Branch):** Відгалужена від головної гілки, вона використовується для інтеграції нових функцій та інших змін. Коли функції готові до випуску, гілка розробки зливається з головною гілкою.
3. **Гілки нових функцій (Feature Branches):** Ці гілки створюються для розробки нових функцій. Вони зазвичай відгалужуються від гілки розробки і зливаються назад у гілку розробки після завершення розробки функції.
4. **Гілки випуску нових версій (Release Branches):** Ці гілки використовуються для підготовки випуску наступної версії продукту (наприклад, для бета-тестування). Вони відгалужуються від гілки розробки та зливаються в головну гілку та назад у гілку розробки, коли випуск завершено.
5. **Гілки гарячих виправлень (Hotfix Branches):** Ці гілки використовуються для швидкого внесення виправлень у випущені версії. Вони відгалужуються від головної гілки та зливаються назад у головну та гілку розробки.
6. **Правила злиття (Merging Rules):** Git Flow чітко регламентує правила для злиття гілок, що допомагає уникнути конфліктів злиття та забезпечує стабільність процесу розробки.
7. **Тегування версій (Version Tagging):** У Git Flow теги створюються в головній гілці, щоб легко можна було ідентифікувати різні версії продукту.

Ця модель сприяє створенню прозорої організації робочого процесу, забезпечує чітке розмежування стадій розробки, тестування та релізу. Однак вона може бути занадто громіздкою і складною в управлінні для маленьких проектів або команд, в таких випадках можуть бути використані альтернативні підходи, такі як GitHub Flow.

**GitHub Flow**. Це спрощена версія Git Flow, адаптована для проектів з постійною інтеграцією (Continuous Integration, CI) і регулярними розгортаннями (Continuous Delivery, CD). Цей робочий процес має такі принципи:

1. Код в основній гілці завжди готовий до розгортання.
2. Нові функції створюються в окремих гілках.
3. Зміни зливаються в основну гілку після перегляду коду та автоматизованого тестування.
4. Як тільки нові зміни злиті виконується розгортання нової версії.
5. Гілки нових функцій таким чином є тимчасовими.

Цей робочий процес оптимально підходить для невеликих команд, проектів з мікросервісною архітектурою і постійно оновлюваних середовищ безперервної доставки.

Вибір робочого процесу із застосуванням системи контролю версій є вирішальним у формуванні процесу розробки. Він впливає на те, як команди співпрацюють, як інтегруються зміни та як розгорається програмне забезпечення і публікується документація. Кожен робочий процес має свої сильні сторони та підходить для різних типів проектів і структур команд. В реальних процесах розробки кожен із процесів може бути модифікований або адаптований для конкретної команди або потреб проекту. Розуміння та вибір правильного робочого процесу є критичним рішенням для ефективної розробки програмного забезпечення, документації та управління проектом, особливо в розподілених командах.

## Інтеграція з сучасними практиками розробки

Інтеграція систем контролю версій з більш іншими практиками розробки є важливим аспектом сучасної програмної інженерії. Ця інтеграція - це не просто технічна реалізація, а дифузія систем контролю версій у культурну та процедурну тканину команд розробників програмного забезпечення. Ефективна інтеграція системи контролю версій з практиками розробки оптимізує співпрацю, підвищує ефективність та забезпечує узгодженість і якість програмних продуктів.

**Гнучкі методології та VCS.** Гнучкі (Agile) методи розробки, що характеризуються ітеративною та інкрементальною розробкою, цілком покладаються на можливості систем контролю версій. Гнучкі методології, такі як Scrum та Kanban, вимагають частих змін кодової бази, роботи з багатьма розподіленими командами, які працюють паралельно над різними новими функціями чи компонентами. Розподілені системи контролю версій роблять можливим такий робочий процес, що дозволяє одночасно виконувати паралельну розробку різних нових функцій, коли члени команд можуть працювати незалежно над окремими гілками, не заважаючи основній кодовій базі [32]. Можливість ефективно об'єднувати ці гілки та попереджати або ефективно вирішувати потенційні конфлікти злиття має вирішальне значення для підтримання гнучкості і ефективності процесу розробки.

**Безперервна інтеграція / безперервне розгортання (CI / CD).** Практики CI / CD є невід'ємною частиною сучасної культури розробки, а системи контролю версій відіграють ключову роль у їх реалізації. Безперервна інтеграція передбачає часту інтеграцію змін коду в спільний репозиторій, що вимагає надійного процесу контролю версій для управління цими змінами [48]. Безперервне розгортання вдосконалює ці процеси за допомогою автоматизованого розгортання усіх нових змін, внесених у кодову базу, після проходження необхідних тестів у інтеграційному середовищі. Системи контролю версій підтримують процедури безперервної інтеграції і розгортання, гарантуючи одночасно цілісність кодової бази і своєчасні оновлення функціоналу програмних продуктів і документації

**Огляд коду та колективна розробка.** Системи контролю версій також полегшують процеси взаємного огляду коду (code review), які є важливими для підтримки якості коду та документації, а також ефективної співпраці між авторами і розробниками. Функції систем, такі як запит на злиття (pull request) у розподілених системах контролю версій дозволяють розробникам пропонувати зміни, які можуть бути переглянуті та обговорені членами команди перед злиттям у основну кодову базу [40]. Цей процес не тільки покращує якість програмного забезпечення, але й заохочує обмін знаннями та ефективне колективне володіння кодом.

**Документація та простежуваність.** Ефективна інтеграція систем контролю версій у практики розробки документації допомагає підтримувати актуальність документації при постійній еволюції кодової бази програмних продуктів. Можливість відстежувати зміни, розуміти історію модифікацій та завжди мати відповідність кожної зміни конкретному авору або розробнику має вирішальне значення як для підзвітності, так і для розуміння обґрунтування певних рішень щодо кодування і документування [33]. Така атрибутивність і простежуваність є неоціненною у складних масштабних проектах, де саме розуміння історії коду може суттєво впливати на зусилля необхідні для підтримки, розгортання, конфігурації та супроводження продуктів.

**Адаптація до сучасних середовищ розробки.** Інтеграція систем контролю версій з практиками розробки також проявляється в її адаптації для підтримки сучасних інтегрованих середовищ розробки (IDE), особливо хмарну розробку та мікросервісну архітектуру. Системи контролю версій еволюціонували, щоб краще підтримувати ці парадигми, пропонуючи функції, такі як розподілений контроль версій, який добре узгоджується з децентралізованою , розподіленою природою цих сучасних архітектур [41].

Таким чином, інтеграція систем контролю версій з сучасними практиками розробки – це багатогранний процес, який охоплює різні аспекти розробки програмного забезпечення і документації. Від сприяння гнучким методологіям до підтримки процесів безперервної інтеграції і розгортання CI / CD, підвищення рівня командної співпраці за допомогою оглядів коду, забезпечення простежуваності та адаптації до сучасних середовищ розробки, системи контролю версій є незамінним інструментом в життєвому циклі розробки програмного забезпечення і документації. У міру того, як практики розробки продовжують еволюціонувати, роль та можливості систем контролю версій, ймовірно, зростуть, ще більше закріплюючи її важливість у галузі програмної інженерії.

## Класифікація систем контролю версій

### За архітектурою

Класифікація систем контролю версій за архітектурою є визначальною для розуміння їхньої еволюції, функціональності та придатності для різних середовищ розробки. Ця класифікація в основному обертається навколо структури та управління сховищами, де зберігаються версіоновані файли. Дві домінуючі архітектурні моделі в системах контролю версій - це централізовані системи контролю версій та розподілені системи контролю версій.

**Централізовані системи контролю версій.** Як випливає з назви, використовують єдиний центральний сервер для зберігання всіх даних та забезпечення спільної роботи команди. Найвідомішим прикладом централізованої системи контролю версій є Subversion (SVN), яка широко застосовувався завдяки своїй простоті і можливості відстежувати зміни ефективніше, ніж її попередники, такі як CVS (Concurrent Versions System) [30]. У централізованих системах кожен користувач отримує свою власну робочу копію, але історія та відстеження версій підтримуються у центральному сховищі. Така архітектура спрощує адміністрування, але має значні недоліки, зокрема єдину точку відмови. Якщо центральний сервер не працює, ніхто не може співпрацювати чи зберігати версіоновані зміни [30].

**Розподілені системи контролю версій**. Характерною особливістю таких систем є децентралізоване, розподілене зберігання версій. Кожен користувач має повну локальну копію репозиторію, включно з усіма файлами та історією змін. Git, створена Лінусом Торвальдсом для розробки ядра Linux, є найвідомішим прикладом розподіленої системи контролю версій. Розподілені системи вирішують обмеження централізованих систем. Вони дозволяють користувачам працювати офлайн і вимагає мережевого доступу лише для синхронізації змін. Крім того, оскільки кожна копія є повноцінним сховищем, це посилює резервування та цілісність даних [35].

Архітектурні відмінності між централізованими та розподіленими системами мають глибокі наслідки для робочого процесу, продуктивності та безпеки. Централізована архітектура часто вважається простішою і зрозумілою, що робить її придатною для менших команд чи проектів з менш частими змінами. З іншого боку, розподілена архітектура пропонує більше гнучкості і краще підходить для масштабних розподілених команд чи проектів, що вимагають високого рівня резервування та потужних можливостей розгалуження і злиття [19].

Підсумовуючи, архітектура системи контролю версій є фундаментальним аспектом, що диктує її операційну динаміку, ефективність та придатність для конкретних проектних середовищ. Перехід від централізованих до розподілених систем протягом років відображає історичну зміну потреб практик розробки програмного забезпечення і документації, зокрема в плані співпраці, масштабованості та цілісності даних.

### За функціональністю та характеристиками

Системи контролю версій можна класифікувати за широким спектром функціональності та характеристик, які вони пропонують. Ця класифікація є важливою для розуміння придатності систем контролю версій для конкретних вимог проекту та робочих процесів. Перелічимо ключові функціональності та характеристики, що відрізняють різні системи контролю версій.

**Можливості розгалуження та злиття**. Однією з основних характеристик, що відрізняють системи контролю версій, є їхній підхід до розгалуження та злиття. Розгалуження дозволяє розробляти кілька версій проекту паралельно, у той час як злиття об’єднує зміни із різних гілок в єдину версію. Розподілені системи керування версіями, такі як Git та Mercurial, пропонують потужні можливості розгалуження та злиття, дозволяючи розпаралелювати та обʼєднювати автономну розробку та ізольовані робочі процеси [35]. Натомість в деяких централізованих системах керування версіями процеси розгалуження та злиття можуть бути більш обмеженими, трудомісткими або менш ефективними.

**Підтримка атомарних операцій**. Атомарні операції – це група операцій, які розглядаються як єдиний блок роботи. Якщо частина операції не вдається, вся операція відміняється, гарантуючи цілісність даних. Системи контролю версій, такі як Subversion, відомі своєю підтримкою атомарних комітів, що є важливим для підтримки узгодженості, особливо у великих проектах [30].

**Механізми вирішення конфліктів**. Вирішення конфліктів – ключова характеристика, особливо в системах, де багато учасників працюють над тими самими файлами. Розподілені системи, такі як Git, забезпечують потужні інструменти для вирішення конфліктів, дозволяючи об'єднувати зміни з різних гілок вручну або автоматично, залежно від складності конфлікту [19].

**Масштабованість**. Здатність системи контролю версій, обробляти великі проекти з численними файлами та учасниками – критично важлива характеристика. Системи, такі як Git, високомасштабовані й здатні ефективно керувати великими проектами [35]. Ця масштабованість часто є вирішальним фактором для великих організацій та проектів з відкритим вихідним кодом.

**Інтерфейс користувача та доступність**. Зручність використання системи контролю версій, включаючи інтерфейс командного рядка (CLI) та графічний інтерфейс користувача (GUI), значною мірою впливає на його поширення. Деякі користувачі віддають перевагу простоті й контролю за допомогою CLI, як у Git, тоді як іншим можуть подобатися більш інтуїтивно зрозумілі GUI, що надаються такими системами контролю версій, як TortoiseSVN чи SourceTree.

**Інтеграція з іншими інструментами**. Масштаб, в якому VCS може інтегруватися з іншими інструментами розробки (IDE, компіляторами, системами зборки, системами безперервної інтеграції та розгортання, системами відстеження помилок, тощо), є життєво важливою характеристикою. Системи, такі як Git і Subversion, пропонують широкі можливості інтеграції, підвищуючи ефективність багатьох робочих процесів життєвого циклу розробки за рахунок автоматизації [33].

**Функції безпеки.** Функції безпеки, включаючи контроль доступу, шифрування та механізми аутентифікації, є важливими для захисту цілісності та конфіденційності коду. Централізовані системи контролю версій, такі як Perforce та Team Foundation Server (TFS), пропонують удосконалені функції безпеки, роблячи їх оптимальними для корпоративних середовищ, де безпека є найважливішим завданням [49].

**Підтримка непрограмних обʼєктів**. Деякі системи контролю версій краще підходять для роботи з бінарними файлами та непрограмними ресурсами, що є важливим у таких галузях, як розробка ігор та графічний дизайн. Наприклад, Perforce та Subversion відомі своєю ефективною роботою з великими бінарними файлами [50].

**Функції автономної роботи**. Розподілені системи контролю версій, такі як Git, дозволяють користувачам працювати офлайн, роблячи коміти та інші операції локально без інтернет з'єднання. Ця функція особливо корисна для учасників, які можуть не завжди мати стабільний доступ до Інтернету [35].

Отже, класифікація систем контролю версій за функціональністю та характеристиками є багатогранним розглядом, що охоплює такі аспекти, як можливості розгалуження та злиття, атомарні операції, вирішення конфліктів, масштабованість, інтерфейс користувача, інтеграцію з іншими інструментами, захист, підтримку непрограмних активів та функції автономної роботи. Вибір системи контролю версій часто залежить від конкретних потреб команди та контексту проекту, де кожна система пропонує унікальну комбінацію цих характеристик.

### За методом доступу

Класифікація систем контролю версій за методом доступу є визначальною для розуміння їх операційної динаміки та придатності для різних команд і середовищ розробки. Методи доступу у системах контролю версій посилаються на способи, якими користувачі взаємодіють з репозиторіями контролю версій, зосереджуючись головним чином на тому, як вносяться, переглядаються та отримуються зміни. Цю класифікацію можна широко категоризувати за трьома основними типами: інтерфейс командного рядка (CLI), графічний інтерфейс користувача (GUI) та інтерфейс на основі Web.

**Інтерфейс командного рядка (CLI)**. Інтерфейс командного рядка є одним з найперших і найтрадиційніших способів взаємодії з системою контролю версій. Системи на основі CLI, такі як Git і Subversion, вимагають від користувачів введення певних команд у термінал або командний рядок для виконання операцій контролю версій [35]. Цей метод дуже звичний для розробників, які віддають перевагу більш гранульованому контролю над операціями своєї системи контролю версій та тими, хто працює в середовищах, де запуск графічних інтерфейсів неможливий або менш ефективними.

CLI пропонує кілька вагомих переваг, включаючи можливість скриптування та автоматизації складних завдань контролю версій, менше навантаження на системні ресурси і часто швидше виконання команд порівняно з системами на основі графічного інтерфейсу користувача [19]. Однак крива навчання може бути надто крутою для непідготовлених користувачів, хто не знайомий з середовищами командного рядка, і це потенційно перешкоджає прийняттю і розповсюдженню серед менш технічно-орієнтованих користувачів.

**Графічний інтерфейс користувача (GUI)**. Інструменти контролю версій з графічним інтерфейсом користувача забезпечують більш зручний для користувача підхід до контролю версій. Інструменти GUI, такі як TortoiseSVN для Subversion або SourceTree для Git, інкапсулюють складність команд систем контролю версій в більш інтуїтивно зрозумілий візуальний інтерфейс [43]. Такий підхід значно знижує бар'єр входу для нових користувачів і є особливо корисним у мультидисциплінарних командах, де не всі члени комфортно почуваються з інттерфейсами командного рядка.

Основною перевагою інтерфейсу на основі GUI є його доступність і простота використання, що робить складні концепції контролю версій більш наочними і зрозумілими для початківців. Однак інструменти GUI можуть не мати деяких передових функцій, доступних у інструментах CLI, і можуть бути більш повільними і вимогливими до ресурсів.

**Web інтерфейс**. Інтерфейси систем контролю версій на основі Web набули популярності з появою онлайн-платформ, таких як GitHub, GitLab та Bitbucket. Ці платформи пропонують веб-інтерфейс для найчастіше використовуваних функцій системи, таких як перегляд репозиторіїв, подання змін, порівняння версій та функції співпраці, такі як запити на злиття та відстеження помилок [40].

Основна перевага Web інтерфейсів полягає в тому, що користувачу не потрібно встановлювати додаткові програмні продукти для того, щоб почати роботу в команді. В сучасних платформах підхід на основі Web також спрощує командну співпрацю, особливо у відкритих та розподілених командах, надаючи інструменти для відстеження помилок (issue tracking), документації (wiki) та огляду коду (code review) в єдиній доступній платформі. Хоча веб-інтерфейси пропонують зручність та функції співпраці, вони часто покладаються на базові інструменти CLI для більш складних операцій і можуть пропонувати неповний набір функціональності, доступний у цих системах.

Підсумовуючи, метод доступу у системах контролю версій є ключовим критерієм класифікації, який впливає на досвід користувача, ефективність співпраці та адаптивність системи до різних робочих середовищ. У той час як CLI пропонує потужність і гнучкість для досвідчених користувачів, інтерфейси GUI та Web інтерфейс забезпечують доступність та функції співпраці, корисні для більш широких команд і менш технічно-орієнтованих користувачів. Вибір методу доступу для системи контролю версій має відповідати рівню майстерності команди, вимогам робочого процесу та складності проектів, які розробляються.

### За рівнем інтеграції

Класифікація систем контролю версій на основі можливостей інтеграції є важливою для розуміння їх адаптивності та ефективності в різних середовищах розробки. Можливості інтеграції відносяться до здатності безперешкодно підключатися та функціонувати з іншими інструментами та системами, включаючи інтегровані середовища розробки (IDE), системи зборки, системи відстеження помилок та системи безперервної інтеграції і безперервного розгортання (CI/CD). Цю класифікацію можна умовно поділити на три рівні: базова інтеграція, розширена інтеграція та повна підтримка DevOps.

**Базова інтеграція**. Цей рівень означає мінімальний необхідний рівень зв'язку з зовнішніми інструментами та системами. Як правило, це включає в себе підтримку популярних IDE, таких як IntelliJ IDEA, Visual Studio Code, Eclipse, де розробники можуть виконувати базові операції контролю версій (коміт, злиття, розгалуження) безпосередньо в інтерфейсі IDE. Базової інтеграції в багатьох випадках буває достатньо для окремих розробників або невеликих команд, щоб підтримувати раціональний робочий процес без постійного перемикання між різними додатками.

Однак базовій інтеграції часто не вистачає повного набору функцій системи контролю версій, таких як розширене вирішення конфліктів, детальне відстеження змін між різними версіями або автоматизовані процеси зборки та тестування. Цей рівень інтеграції підходить для простіших проектів з менш складними вимогами до розробки та розгортання.

**Розширена інтеграція.** Цей більш поглиблений рівень інтеграцій означає що система контролю версій не тільки інтегрується в локальне середовище розробки (IDE) користувача, але й взаємодіє з іншими інструментами розробки, такими як системи перевірки коду (compilers, linters), програмне забезпечення для відстеження помилок (bug trackers) та сервери збoрки (build systems). Наприклад, інтеграція Git з такими інструментами, як Jenkins для безперервної інтеграції або JIRA для відстеження помилок, демонструє цей рівень.

Такий рівень інтеграції дозволяє оптимізувати робочий процес, даючи можливість для кращої автоматизації процесів, таких як запуск зборки при злитті нових змін або встановлення відповідності змін коду з конкретними помилками у системі відстеження помилок. Розширена інтеграція особливо корисна для проектів середнього та великого масштабу, де відстеження прогресу та забезпечення якості коду стає складнішим.

**Повна підтримка DevOps**. Це найвищий можливий рівень інтеграції систем контролю версій, де система контролю версій є невід'ємною частиною комплексного підходу DevOps, що охоплює безперервну інтеграцію, безперервне розгортання, моніторинг та зворотній зв'язок. Системи контролю версій, такі як Git, при використанні разом з платформами GitHub або GitLab, пропонують цей рівень інтеграції, надаючи інструменти для CI/CD, підтримку контейнеризації (наприклад Docker) та навіть моніторинг і аналітику [51]. На практиці це означає, що не тільки вихідний код програмного продукту, але й усі конфігурації, скрипти автоматизації, що керують перевіркою, зборкою та розгортанням проектів і всі їхні версії також зберігаються в системі контролю версій. Саме файли в системі контролю версій вважаються єдиним джерелом істини (single source of truth). При цьому будь-яка зміна, наприклад встановлення нової версії компонента для тестування або інтеграції, здійснюється виключно шляхом редагування відповідного файлу конфігурації в системі контролю версії. Такий підхід до управління конфігурацією називається GitOps [52]. Переваги такого підходу в тому, що будь-який стан системи завжди є описаний в текстовому вигляді, придатному для читання і порівняння в будь-який момент, зміни в конфігурацію вносяться через процедуру перегляду коду, таким чином інші учасники проекту завжди знають про зміни, а також в тому, що авторство будь-якої зміни може простежити будь-який учасник проекту, що позитивно впливає на відповідальність при здійснені таких змін.

Цей рівень інтеграції є важливим для гнучких середовищ розробки з високим темпом внесення змін, де швидке розгортання нових версій та часті ітерації є нагальною потребою. Він підтримує культуру безперервного вдосконалення та сприяє високому ступеню автоматизації, значно зменшуючи стихійний вплив людського фактору і кількість випадкових помилок при неавтоматизованих операціях та підвищуючи загальну ефективність і взаємодію між командами розробки, тестування та експлуатації.

Отже, можливості інтеграції Системи Контролю Версій є важливим критерієм класифікації, який значно впливає на її ефективність в різних проектних середовищах. У той час як базової інтеграції достатньо для простіших проектів, розширена інтеграція та повна підтримка DevOps є необхідними для більш складних, динамічних та кооперативних середовищ розробки. Тому вибір системи контролю версій повинен бути узгоджений зі складністю проекту, розміром команди та бажаним рівнем автоматизації і інтеграції з іншими інструментами та системами.

### За галузевою специфікою використання

Сучасне застосування систем контролю версій виходить далеко за рамки сфери розробки програмного забезпечення, проникаючи в різні галузі, де відстеження змін, командна співпраця, координація колективних зусиль та підтримка історичних записів мають вирішальне значення. Розглянемо використання систем контролю версій в різних галузях, висвітлюючи унікальні галузеві вимоги та потрібні для них адаптації.

**Розробка програмного забезпечення та ІТ-послуги**. Найбільш поширене використання систем контролю версій традиційно спостерігається в галузі розробки програмного забезпечення та ІТ-послуг. Тут контроль версій є критично необхідним протягом усього життєвого циклу розробки програмних продуктів для управління вихідним кодом, колективної співпраці, відстеження версій, мінімізації конфліктів під час розробки [35]. Системи контролю версій є необхідними для реалізації практик DevOps, таких як GitOps, дозволяючи безперервну інтеграцію та доставку шляхом підтримки цілісності коду та автоматизації процесів розгортання [53].

**Академічні дослідження та освіта**. В академічних колах системи контролю версій дедалі частіше застосовуються для управління дослідницькими проектами, особливо тих, що застосовують обчислювальні та статистичні методи. Системи контролю версій допомагають у відстеженні змін в дослідницьких даних, сценаріях та рукописах, сприяючи відтворюваності досліджень та прозорості всього процесу досліджень [54]. В освіті Git використовується для подання завдань, версіювання навчальних матеріалів та сприяння спільним освітнім проектам [54].

**Творчі галузі**. У творчих галузях, таких як графічний дизайн, відеовиробництво та веб-розробка, системи контролю версій мають бути адаптовані для роботи з нетекстовими файлами, такими як зображення та мультимедіа. Такі системи, як Adobe Version Cue та Git Large File Storage (LFS), призначені для управління великими бінарними файлами, що зазвичай використовуються для зберігання відео, забезпечуючи при цьому контроль версій, і оптимізуючи зберігання та копіювання даних.

**Виробництво та інжиніринг**. Контроль версій у виробництві та інжинірингу має вирішальне значення для управління конструкціями виробів, файлами САПР та інженерними кресленнями. Системи контролю версій в цих секторах також повинні підтримувати великі бінарні файли і інтегруватися з програмним забезпеченням автоматизованого проектування (САПР). Такі системи, як Git LFS і SVN, були адаптовані для задоволення цих потреб, забезпечуючи точне версіювання файлів креслень конструкцій та сприяючи співпраці між інженерними групами.

**Охорона здоров'я та фармацевтична промисловість**. В охороні здоров'я та фармацевтиці системи контролю версій використовуються для управління документацією, дослідницькими даними та регуляторними поданнями. Акцент в цих галузях робиться на дотриманні правових та регуляторних стандартів, таких як HIPAA [56] в охороні здоров'я та GxP [57] у фармацевтиці. Системи контролю версій у цих секторах часто мають посилені заходи безпеки та аудиторські записи для забезпечення цілісності даних та дотримання вимог [58].

**Державний та громадський сектори**. Урядові установи можуть використовувати системи контролю версій для управління державними записами, юридичними документами та процесами розробки політики. Такі системи могли б забезпечувати високий рівень безпеки, підзвітності та прозорості [59], а також допомагали бы уникати помилок . Відкриті системи, такі як Git, мають перевагу в цьому секторі завдяки своїй прозорості та пристосованості до потреб громадського сектора.

Підсумовючи, різноманітне застосування систем контролю версій в різних галузях підкреслює універсальність систем контролю версій та їх важливу роль в управлінні цифровими активами. Хоча основні принципи систем контролю версій залишаються однаковими, галузеві адаптації мають вирішальне значення для вирішення унікальних проблем, таких як типи файлів, дотримання нормативних та регуляторних вимог і специфічних потреб у співпраці.

### Еволюційні тенденції в класифікації

Класифікація систем контролю версій значно еволюціонувала з часом, відображаючи динамічну природу практик розробки програмного забезпечення та зростаючу складність управління проектами. Ця еволюція – це не просто поява нових систем, а й трансформація існуючих систем для адаптації до нових вимог і технологічних досягнень. Розглянемо ключові еволюційні тенденції в класифікації систем контролю версій, зосередившись на тому, як ці тенденції сформували сучасний ландшафт контролю версій.

**Виникнення гібридних систем**. Спочатку системи контролю версій класифікувалися виключно як централізовані або розподілені системи. Однак останні тенденції розвитку вказують на розмиття цих меж, що призводить до появи гібридних систем. Такі системи мають на меті поєднати переваги як централізованих, так і розподілених моделей, пропонуючи гнучкість у робочих процесах та підвищену ефективність у керуванні великими кодовими базами. Наприклад, деякі сучасні системи дозволяють централізовано керувати певними аспектами сховища, водночас підтримуючи розподілені операції для інших.

**Посилений акцент на інтеграцію та автоматизацію**. Класифікація систем контролю версій також еволюціонувала в напрямку акцентування можливостей інтеграції, зокрема з інструментами безперервної інтеграції / безперервного розгортання (CI/CD) та іншими автоматизованими процесами розробки програмного забезпечення [53]. Ця тенденція відображає зростаюче значення практик DevOps у розробці ПЗ, де безшовна інтеграція контролю версій з автоматизованими збірками та системами тестування має вирішальне значення.

**Спеціалізація та диверсифікація**. Ще однією помітною тенденцією є спеціалізація та диверсифікація систем контролю версій для задоволення конкретних потреб або галузей. Наприклад, існують системи, спеціально призначені для керування великими бінарними файлами, такі як Git Large File Storage (LFS) - розширення Git [53]. Аналогічно, деякі системи контролю версій адаптуються під конкретні мови програмування чи фреймворки, пропонуючи розширені функціональності для спеціалізованих потреб і форматів зберігання даних.

**Хмарні та web-інтегровані системи**. Підйом хмарних обчислень також вплинув на класифікацію систем контролю версій. Сучасні систем контролю версій все частіше пропонують хмарий хостинг та web-інтеграцію, забезпечуючи переваги з точки зору доступності, співпраці та резервного копіювання. Платформи як GitHub та GitLab уособлюють цю тенденцію, поєднуючи функціональність систем контролю версій з веб-інтерфейсом для управління проектами, відстеження помилок та документації.

**Орієнтація на користувача та функції співпраці**. Сучасна класифікація систем контролю версій також відображає зрушення в бік більш орієнтованих на користувача та колаборативних функцій. Це включає в себе вдосконалені інтерфейси користувача, інструменти співпраці в режимі реального часу та функції, які підтримують нетехнічних учасників програмного проекту [33]. Акцент робиться на тому, щоб зробити контроль версій більш доступним та інклюзивним, орієнтуючись на ширше коло користувачів, а не лише розробників програмного забезпечення.

Підсумовуючи, класифікація систем контролю версій еволюціонувала у відповідь на зміни у практиках розробки, технологічні досягнення та потреби користувачів. Ця еволюція призвела до появи більш досконалих, інтегрованих та орієнтованих на користувача систем, що вказує на тенденцію до створення більш універсальних і всеосяжних інструментів контролю версій. Оскільки розробка продовжує еволюціонувати, ймовірно, що класифікація систем контролю версій продовжуватиме адаптуватися, відображаючи нові виклики та можливості у цій галузі.

# Список використаних джерел

1. Офіційний вебпортал парламенту України: [Інтернет-портал]. URL: https://zakon.rada.gov.ua/laws/show/1306-2001-%D0%BF#Text (дата звернення: 18.08.2023).
2. Розетський камінь // Українська Вікіпедія : [Веб-сайт]. URL: https://uk.wikipedia.org/wiki/Розетський\_камінь (дата звернення: 14.10.2023).
3. Поліглотта // Українська Вікіпедія : [Веб-сайт]. URL: https://uk.wikipedia.org/wiki/Поліглотта (дата звернення: 14.10.2023).
4. ДСТУ 7564:2014. Інформаційні технології. Криптографічний захист інформації. Функція гешування.. Київ, 2015. 39 с.
5. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. ISO/IEC, 2011.
6. Recommended composition of influenza virus vaccines for use in the 2024 southern hemisphere influenza season // Всесвітня організація охорони здоров'я: [Веб-сайт]. URL: https://www.who.int/publications/m/item/recommended-composition-of-influenza-virus-vaccines-for-use-in-the-2024-southern-hemisphere-influenza-season (дата звернення: 14.10.2023).
7. Guideline for the pharmacological treatment of hypertension in adults // Всесвітня організація охорони здоров'я: [Веб-сайт]. URL: https://iris.who.int/bitstream/handle/10665/344424/9789240033986-eng.pdf (дата звернення: 14.10.2023).
8. Населення // Державна служба статистики України: [Веб-сайт]. URL: https://www.ukrstat.gov.ua/operativ/operativ2007/ds/nas\_rik/nas\_u/nas\_rik\_u.html (дата звернення: 14.10.2023).
9. The top 10 causes of death // Всесвітня організація охорони здоров'я: [Веб-сайт]. 2020. URL: https://www.who.int/news-room/fact-sheets/detail/the-top-10-causes-of-death (дата звернення: 14.10.2023).
10. Wuchty, S., Jones, B. F., & Uzzi, B. (2007). The Increasing Dominance of Teams in Production of Knowledge. Science, 316(5827), 1036–1039. doi:10.1126/science.1136099
11. ISO/IEC/IEEE 26511:2018 Systems and software engineering — Requirements for managers of information for users of systems, software, and services. ISO/IEC/IEEE 2018
12. Morgan K. Technical Writing Process: The simple, five-step guide that anyone can use to create technical documents such as user guides, manuals, and procedures. Better On Paper Publications, 2015.
13. Hadjerrouit, S. Wiki as a collaborative writing tool in teacher education: Evaluation and suggestions for effective use. Computers in Human Behavior. 2014, №32, 301-312.
14. Wichmann A. , Rummel N. Improving revision in wiki-based writing: Coordination pays off // Computers & Education. 2013.№ 62. 262-270.
15. Tichy, Walter F. RCS – A System for Version Control. In: Software: Practice and Experience. 1985. Vol 15. № 7. 637–654.
16. Nagel W. Subversion Version Control: Using the Subversion Version Control System in Development Projects. Prentice Hall, 2005.
17. Thomas D. , Hunt A. Pragmatic Version Control using CVS. The Pragmatic Bookshelf, 2003.
18. Wingerd L. Practical Perforce. O'Reilly Media, 2005.
19. Loeliger, J., McCullough, M. Powerful tools and techniques for collaborative software development. O'Reilly Media, 2012.
20. O'Sullivan B. Mercurial: the definitive guide. O'Reilly Media, 2009.
21. Gyerik J. Bazaar Version Control: A fast-paced practical guide to version control using Bazaar. Packt Publishing, 2013.
22. Roundy D. Darcs: distributed version management in Haskell // Proceedings of the 2005 ACM SIGPLAN workshop on Haskell. Tallinn, Estonia, 2005. 1-4.
23. AlShebli, B.K., Rahwan, T. & Woon, W.L. The preeminence of ethnic diversity in scientific collaboration. Nature Communications 9, 5163 (2018). <https://doi.org/10.1038/s41467-018-07634-8>
24. Yunker J. Beyond borders: Web globalization strategies. Indianapolis, Ind: New Riders, 2003.
25. Localization vs. Internationalization // The World Wide Web Consortium: [Веб-сайт]. 2005. URL: https://www.w3.org/International/questions/qa-i18n (дата звернення: 14.10.2023).
26. Weiss E. H. How to write usable user documentation. Phoenix, AZ: Oryx Press, 1991.
27. Blakiston R. Usability Testing: A Practical Guide for Librarians. Rowman & Littlefield Publishers, 2014.
28. Markel M. , Selber S. A. Technical Communication, Twelfth Edition. Bedford/St. Martin's, 2017.
29. Rochkind, M. J. The source code control system. IEEE Transactions on Software Engineering, SE-1(4), 364–370. (1975)
30. Collins-Sussman, B., Fitzpatrick, B. W., Pilato, C. M. Version Control with Subversion. O'Reilly Media, 2004.
31. Babich, W. A. Software Configuration Management: Coordination for Team Productivity. Addison-Wesley, 1986.
32. Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000.
33. Spinellis, D. Version Control Systems. IEEE Software, 22(5), 108–109, 2005
34. Tammaro, S. G., Mosier, J. N., Goodwin, N. C., Spitz, G. Collaborative Writing Is Hard to Support. A Field Study of Collaborative Writing. Computer Supported Cooperative Work (CSCW), 6(1), 19–51, 1997.
35. Chacon, S., Straub, B. Pro Git. Apress, 2014
36. Driessen, V. A successful Git branching model // Nvie.com: [Веб-сайт]. 2010. URL: <https://nvie.com/posts/a-successful-git-branching-model> (дата звернення: 20.11.2023).
37. Jabbari, R., Bin Ali, N., Petersen, K.,Tanveer, B. What is DevOps? A Systematic Mapping Study on Definitions and Practices. Scientific Research Publishing, 2016.
38. Caram, F. L., Rodrigues, B. R. D. O., Campanelli, A. S., Parreiras, F. S. Machine Learning Techniques for Code Smells Detection. A Systematic Mapping Study. International Journal of Software Engineering and Knowledge Engineering, 29(02), 285–316, 2019.
39. Almorsy, M., Grundy, J., Müller, I. An Analysis of the Cloud Computing Security Problem, 2016.
40. Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J. Social coding in GitHub. Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work - CSCW, 2012
41. Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L. Microservices: Yesterday, Today, and Tomorrow. Present and Ulterior Software Engineering, 195–216, 2017
42. Fogel, K. Producing Open Source Software: How to Run a Successful Free Software Project. O'Reilly Media, 2005
43. Swicegood, T. Pragmatic Version Control Using Git. Pragmatic Bookshelf, 2008
44. Mens, T. A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering, 28(5), 449–462, 2002
45. Андрєєв А. Використання систем контролю версій для колективної роботи над мультимедійною документацією // Github.com: [Веб-сайт]. 2023. URL: https://github.com/ludenus/uad-multimedia-doc (дата звернення: 20.11.2023).
46. Vesperman, J. Essential CVS. O'Reilly Media, 2006
47. Bird, C., Zimmermann, T. Assessing the value of branches with what-if analysis. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE, 2012.
48. Fowler, M. Continuous Integration // Martinfowler.com: [Веб-сайт]. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (дата звернення: 21.11.2023).
49. Berczuk, S., Appleton, B. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Addison-Wesley, 2003.
50. Cohen, R. Managing Software Development with Trac and Subversion. Packt Publishing, 2005.
51. Shahin, M., Ali Babar, M., Zhu, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access, 5, 3909–3943, 2017.
52. GitOps // en.wikipedia.org: [Веб-сайт]. URL: https://en.wikipedia.org/wiki/DevOps#GitOps (дата звернення: 22.11.2023).
53. Humble, J., Farley, D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional, 2010.
54. Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. V., Vizcaíno, J. A. Ten simple rules for taking advantage of Git and GitHub. PLOS Computational Biology, 12(7), 2016.
55. Bryan, J. Excuse Me, Do You Have a Moment to Talk About Version Control? The American Statistician, 72(1), 20–27, 2017.
56. HEALTH INSURANCE PORTABILITY AND ACCOUNTABILITY ACT OF 1996 // www.govinfo.gov: [Веб-сайт]. URL: https://www.govinfo.gov/content/pkg/PLAW-104publ191/html/PLAW-104publ191.htm (дата звернення: 22.11.2023).
57. GxP // en.wikipedia.org: [Веб-сайт]. URL: https://en.wikipedia.org/wiki/GxP (дата звернення: 22.11.2023).
58. Steinwandter, V., Herwig, C. Provable Data Integrity in the Pharmaceutical Industry based on Version Control Systems and the Blockchain. PDA Journal of Pharmaceutical Science and Technology, 2019.
59. On Legislative Collaboration and Version Control by John Wonderlich // Sunlightfoundation.com: [Веб-сайт]. 2012. URL: https://sunlightfoundation.com/2012/09/27/on-legislative-collaboration-and-version-control/ (дата звернення: 22.11.2023).
60. Why Can’t We Have a Version Control System for the Law? // www.laserfiche.com: [Веб-сайт]. URL: https://www.laserfiche.com/resources/blog/why-cant-we-have-version-control-system-law/ (дата звернення: 22.11.2023).