МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія (кафедра) комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ВПРОВАДЖЕННЯ СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ В ПРОЦЕС ФРОНТЕНД РОЗРОБКИ

Виконав: студент групи 2П-21
Спеціальності
121 Інженерія програмного забезпечення
Богдан ДЗЮБА
Керівник:
Станіслав МАРЧЕНКО

Черкаси 2025

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

Кафедра комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри КІ та ІТ

_____ Владислав ХОТУНОВ
(підпис)
«_____»_____2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Дзюбі Богдану Сергійовичу

1. Тема кваліфікаційної роботи Впровадженя системи контролю версій в процес фронтенд розробки

Керівник роботи Марченко Станіслав Віталійович, спеціаліст І категорії
затверджені наказом закладу вищої освіти від «07» жовтня 2024 року № 68у.
2. Строк подання студентом кваліфікаційної роботи 02.06.2025
3. Вихідні дані до кваліфікаційної роботи система контролю версій Git, платформи GitHub для організації репозиторіїв, інструменти GitHub Actions для автоматизації CI/CD-процесів
4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити) огляд предметної області (основи роботи систем контролю версій, їх роль у фронтенд розробці, аналіз популярних рішень, типові проблеми командної розробки без СКВ), проєктування та впровадження системи контролю версій (аналіз вимог до організації репозиторію, вибір інструментів, розробка workflow, створення шаблонів гілок і комітів, інтеграція з CI/CD), практична реалізація (налаштування репозиторію фронтенд застосунку, впровадження системи Git, створення та тестування розробницького процесу, оцінка ефективності впровадженого рішення).
5. Дата видачі завдання 15.09.2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання з підписами керівника і студента
1	Вступ	14.10.2024	
2	Розділ 1. Теоретичні основи систем контролю версій	09.12.2024	
3	Розділ 2. Особливості процесу розроблення фронтенду	10.03.2025	
4	Розділ 3. Впровадження системи контролю версій у розроблення фронтенду	28.04.2025	
5	Висновки	12.05.2025	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2025	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2025	
8	Подання кваліфікаційної роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	10.06.2025	

Студент _____ Богдан ДЗЮБА
(підпис)

Керівник роботи _____ Станіслав МАРЧЕНКО
(підпис)

АНОТАЦІЯ

Кваліфікаційна робота присвячена дослідженню та впровадженню системи контролю версій у процес фронтенд розробки вебзастосунків. У межах роботи проаналізовано роль систем контролю версій у забезпеченні ефективної командної роботи, збереженні історії змін, управлінні релізами та автоматизації процесів розгортання.

Особливу увагу приділено практичному аспекту впровадження Git як найпоширенішої системи контролю версій у сучасних frontend-проєктах. Проведено порівняння процесів розробки з використанням системи контролю версій та без неї, виявлено ключові переваги: зменшення кількості помилок, підвищення прозорості змін, спрощення інтеграції нового функціоналу.

У роботі також описано структуру організації репозиторіїв, створення гілок, застосування стратегій Git Flow, налаштування CI/CD (безперервної інтеграції та доставки), використання GitHub Actions та інших інструментів для автоматизації.

Розроблено методологію впровадження системи контролю версій, що включає послідовні етапи адаптації у команді, технічні рекомендації та аналіз результатів впровадження.

Результати дослідження можуть бути корисними для команд розробників, менеджерів проєктів і компаній, які прагнуть підвищити якість і керованість процесу фронтенд розробки.

Ключові слова: СИСТЕМА КОНТРОЛЮ ВЕРСІЙ, GIT, ФРОНТЕНД, РОЗРОБКА, CI/CD, АВТОМАТИЗАЦІЯ, КОМАНДНА РОБОТА.

ABSTRACT

The qualification thesis is devoted to the research and implementation of a version control system in the process of frontend web application development. The study analyzes the role of version control systems in ensuring effective teamwork, preserving change history, managing releases, and automating deployment processes.

Particular attention is paid to the practical implementation of Git as the most widely used version control system in modern frontend projects. A comparison of development workflows with and without version control is conducted, highlighting key advantages such as error reduction, improved change traceability, and simplified integration of new functionality.

The thesis also describes the structure of repository organization, branch management, application of Git Flow strategies, configuration of CI/CD (Continuous Integration and Delivery), and the use of GitHub Actions and other automation tools.

A methodology for implementing a version control system is developed, including sequential adaptation stages for the team, technical recommendations, and an analysis of implementation results.

The findings of this research can be useful for development teams, project managers, and companies aiming to improve the quality and manageability of the frontend development process.

Keywords: VERSION CONTROL SYSTEM, GIT, FRONTEND, DEVELOPMENT, CI/CD, AUTOMATION, TEAMWORK.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

СКВ – система контролю версій

NPM – Node Package Manager, менеджер пакетов для середовища виконання Node.js

SCSS – Sassy Cascading Style Sheets

RCS – Revision Control System

SVN – Subversion

CVS – Concurrent Versions System

Git LFS – Large File Storage

ВСТУП

Зі збільшенням складності вебзастосунків та мобільних додатків, зростанням кількості пристроїв з різними характеристиками дисплеїв, а також підвищенням вимог до швидкодії та зручності користувацьких інтерфейсів, процес розроблення фронтенду стає все більш комплексним та багатогранним. У сучасних умовах над одним проєктом часто працює команда розробників, що знаходяться в різних географічних локаціях. Це також створює низку викликів, пов’язаних з координацією роботи, уникненням конфліктів у коді, забезпеченням якості та керованості процесу розробки. Крім того, фронтенд-розробка характеризується швидкою зміною технологій, фреймворків та бібліотек, що потребує постійної адаптації робочих процесів.

У цьому контексті системи контролю версій (СКВ) стають не просто корисним інструментом, а необхідним складником ефективного процесу фронтенд-розробки. Вони дозволяють відстежувати зміни у коді, організовувати паралельну роботу над різними функціями, безпечно експериментувати з новими підходами, а також забезпечувати стабільність кінцевого продукту.

Незважаючи на широке розповсюдження систем контролю версій у галузі розробки програмного забезпечення в цілому, їх впровадження у процес фронтенд-розробки має свої особливості та виклики. Специфіка таких проєктів, часто передбачає включення не лише програмного коду, але й великої кількості ресурсів (зображення, шрифти, стилі), а також особливості процесів тестування, розгортання та взаємодії з бекенд-частиною застосунків.

Актуальність даної теми підтверджується статистикою: згідно з дослідженням Stack Overflow Developer Survey 2024, понад 65,000 розробників взяли участь в щорічному опитуванні про кодування, технології та інструменти, які вони використовують [1]. Дослідження State of Frontend 2024 [2] показує, що GitHub має переважаючу популярність серед розробників з 77.9% голосів, що пояснюється його широкою функціональністю та зручністю в безкоштовній

версії, а також фактичним стандартом у промисловому виробництві програмного забезпечення.

Статистика також показує зростаючий ринок систем контролю версій: у 2024 році глобальний ринок СКВ був оцінений у $708.46 мільйонів і за прогнозами має досягти $761.1 мільйона у 2025 році, що обумовлено впровадженням хмарних технологій та інструментів розробки на основі ШІ [3]. Звідси, дослідження процесу впровадження систем контролю версій у хід розроблення фронтенду є актуальним та має як теоретичне, так і практичне значення для підвищення ефективності створення вебзастосунків та мобільних додатків.

Об’єктом дослідження виступають системи контролю версій та керування репозиторіями програмного коду на основі їх інструментів.

Предметом дослідження є методологічні та практичні аспекти впровадження систем контролю версій у процес фронтенд-розробки.

Мета дослідження полягає у розробці комплексного підходу до впровадження системи контролю версій у процес розроблення фронтенду, що орієнтовно дозволить підвищити ефективність роботи команди розробників, забезпечити вищу якість коду та прискорити процес випуску продукту. Відповідно до поставленої мети, було визначено такі завдання дослідження**:**

1. Проаналізувати теоретичні основи функціонування систем контролю версій, дослідити особливості процесу розроблення фронтенду та визначити ключові вимоги до систем контролю версій у цьому контексті.
2. Провести порівняльний аналіз поширених систем контролю версій з точки зору їх застосування у фронтенд-розробці.
3. Розробити методологію впровадження системи контролю версій у процес розроблення фронтенд розробки, визначити оптимальні підходи до налаштування робочого процесу (workflow) з використанням системи контролю версій для фронтенд-проєктів.
4. Практично впровадити розроблену методологію на прикладі експериментального проєкту.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ СИСТЕМ КОНТРОЛЮ ВЕРСІЙ

1.1 Поняття та класифікація систем контролю версій

Система контролю версій – це програмний інструмент, який фіксує зміни у файлі чи наборі файлів протягом певного часу, дозволяючи за потреби повернутися до будь-якої попередньої версії [4]. З її допомогою розробники можуть відстежувати повну історію змін (з авторами, датами, коментарями) й уникати конфліктів під час одночасної роботи над кодом [5]. Наприклад, СКВ дає можливість в будь-який момент відновити стан проєкту на певній ревізії або версії, скасувати небажані зміни тощо [5].

Першою СКВ вважається SCCS, розроблена Марком Рокіндом у 1972 році для системи IBM System/370[6]. Вона зберігала зміни файлів як дельти і мала обмеження: кожен файл версіювався окремо, без можливості атомарних комітів кількох файлів [6]. У 1982 році Уолтер Тіши створив систему RCS, яка суттєво вплинула на подальший розвиток СКВ. Вона аналогічно зберігала лише відмінності (дельти) між версіями, але використовувала «reverse deltas» і дозволяла відтворювати будь-яку ревізію файлу [4,6]. Варто зазначити, що RCS досі входить до складу багатьох UNIX-систем [4].

Наступним етапом стали централізовані СКВ. Наприкінці 1980-х і в 1990-х роках з’явилися CVS (Concurrent Versions System) та Subversion (SVN), які вперше надали спільний сервер – єдине сховище для всіх файлів проєкту [4]. Наприклад, Subversion, створений компанією CollabNet у 2000 році, став поширеною заміною CVS [4]. У централізованих системах усі розробники працюють з одним «офіційним» репозиторієм на сервері: це полегшує адміністрування й обмін інформацією, але створює єдину точку відмови, якщо сервер недоступний, жодних змін зробити не можна [4].

У 2005 році Лінус Торвальдс запропонував принципово новий підхід – розподілені СКВ. Так з’явилися Git і Mercurial. У таких системах кожен розробник має повну копію сховища (репозиторія) з усією історією на

локальному комп’ютері. Це дозволяє робити коміти і гілкування офлайн, а синхронізація відбувається через передавання змін між репозиторіями (push/pull) [7]. Головна перевага РСКВ – відсутність єдиної точки відмови: якщо центральний сервер зникає, будь-який клон може відновити репозиторій [4]. До того ж розподілені системи підтримують гнучкі схеми співпраці (наприклад, кілька віддалених гілок чи ієрархічні робочі процеси – workflow), які неможливі в централізованих системах [4].

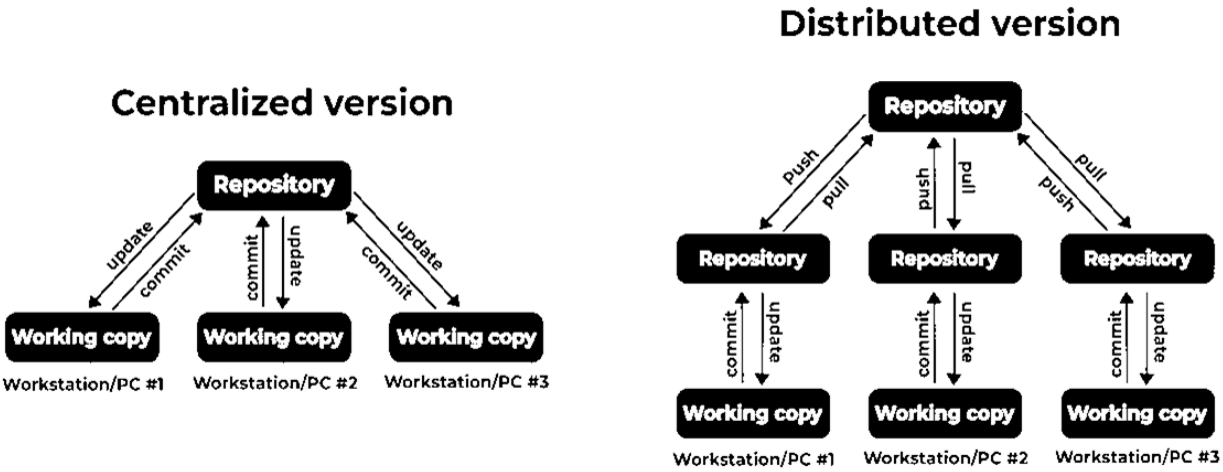


Рисунок 1.1 – Візуалізація роботи а) централізованих СКВ; б) розподілених СКВ [8]

Разом з розвитком Git з’явилися й сервіси хостингу репозиторіїв. Наприклад, GitHub було запущено у квітні 2008 року як вебплатформу для спільної роботи над Git-проєктами. З того часу GitHub (та подібні сервіси) зробили розподілені системи найпопулярнішим вибором для розробки, особливо у веб- та фронтенд-проєктах [9] Звідси, розвиток СКВ пройшов три основні етапи: локальні системи для одного користувача, централізовані системи з сервером, і розподілені системи з повними клонуваннями.

На основі цього пропонують різні способи класифікувати системи контролю версій. Наприклад, за архітектурою, як уже було вказано, СКВ можуть бути локальні, централізовані та розподілені. У той же час, за зберіганням змін СКВ можуть базуватись на файловій, дельтовій чи знімковій моделях. При

використанні файлової (або кодувальна) моделі СКВ зберігає для кожного файлу окремий запис його версій. Наприклад, RCS створює спеціальні файли-історії (s-файли), у яких послідовно зберігаються різниці між версіями. Такі системи зручно відтворюють будь-яку ревізію окремого файлу, але атомарних комітів для набору файлів не підтримують [7].

Дельтова модель передбачає, що СКВ зберігає тільки зміни (дельти) від однієї ревізії до іншої. Таке рішення використовували SCCS і RCS: у них історія зберігається у вигляді набору патчів (дельт). Такий підхід економить місце, але для відновлення старої версії потрібно послідовно застосувати всі дельти.

Знімкова (snapshot) модель, характерна для Git, базується на тому, що робляться знімки усього дерева файлів при кожному коміті. Тобто в репозиторії зберігається «фотографія» стану проєкту в кожному моменті. Якщо файл не змінюється, Git зберігає лише посилання на попередню копію. Така модель спрощує доступ до повного стану проєкту в будь-який момент і швидко виконує операції (особливо локальні), хоч може займати більше місця без ретельної оптимізації.

За робочим процесом (workflow) можна класифікувати системи контролю версій:

1. Лінійний (trunk-based) workflow. При цьому кожен розробник комітить зміни безпосередньо у єдину головну гілку (наприклад, trunk в SVN або master в Git) [5]. Це спрощений підхід («як у Google Docs»), але ускладнює керування статусом завдань і тестування, оскільки часто робиться більше комітів прямо у основну гілку [5]. Такий стиль був традиційним для SVN-депо.

2. Зі злиттям (branching/merging). Більшість сучасних СКВ (особливо Git) розроблені з урахуванням активного гілкування. Розробники створюють окремі гілки для нових функціональних можливостей (features) чи виправлень (fix), а потім зливають (merge) їх в основну гілку. За рахунок цього робочий процес стає нелінійним та гнучким. Git, зокрема, має потужні механізми створення та злиття гілок, що роблять цей процес інтуїтивно зрозумілим і швидкий [5, 10].

У порівнянні зі старими системами, у Git/Mercurial гілкування більш «легке»: можна створити нову гілку і переключитися на неї буквально за секунду. Ці класифікації демонструють різні підходи до побудови СКВ. Наприклад, Git – це розподілена система, яка зберігає знімки всієї кодової бази, підвищуючи швидкість локальних операцій [4]. У той самий час, SVN – централізована система з дельтовим зберіганням, де всі коміти відправляються на сервер [10]. Вибір моделі залежить від вимог проєкту: чи потрібна масштабованість та офлайн-робота (Git/Mercurial), чи важливо централізоване адміністрування та стабільність (SVN/Perforce).

1.2 Основні функції систем контролю версій

Однією з базових функціональних можливостей систем контролю версій є відстеження історії змін. СКВ фіксує кожен коміт, зберігаючи авторів, дати і коментарі до змін. Таким чином формується повна історія проєкту. Це дозволяє аналізувати, хто і коли вніс конкретні зміни, та відкотити код до будь-якої попередньої версії [5]. Наприклад, у логах видно, коли було виправлено баг або коли додано певну функцію.

Типовою функцією також є скасування (undo) змін. За потреби СКВ може повернути окремі файли або весь проєкт у попередній стан [4]. Якщо нові зміни призвели до помилки, розробник може швидко відкотитися до відомо працездатної ревізії. Це підвищує надійність розробки: випадково зіпсований код легко відновити.

Підтримка гілок (branches) дозволяє паралельно працювати над різними версіями проєкту [5]. Розробник може створити окрему гілку для нової функції чи експерименту, не впливаючи на основну гілку. Після завершення робіт гілка зливається (merge) з головною гілкою, інтегруючи зміни. У сучасних СКВ (особливо Git) мережі галуження/злиття дуже гнучкі – можна легко працювати з великою кількістю гілок без складної конфігурації [5].

Функціонально доступними в сучасних СКВ є теги – позначки на конкретних комітах, що зручно використовувати для міток версій чи релізів. За допомогою тегів можна швидко позначати випуск нової версії програмного забезпечення й повертатися до нього в майбутньому. Наприклад, «v1.0.0» може бути тегом на коміті, що відповідає випуску першої стабільної версії.

У розподілених СКВ передбачено механізми обміну даними між репозиторіями. Команда push відправляє локальні коміти на віддалений сервер або інший репозиторій, а pull отримує свіжі зміни назад [5]. Завдяки цьому розробники можуть легко обмінюватися кодом через мережу або Інтернет. Синхронізацію можна налаштувати через різні протоколи (SSH, HTTP), що дозволяє працювати у будь-яких середовищах.

Кожен коміт у СКВ зазвичай розглядається як атомарна операція: всі зміни включаються в одну транзакцію. Атомарність гарантує, що або всі файли фіксуються (commit) разом, або жодний. Це унеможливлює ситуацію, коли частина змін застосована, а частина – ні, що могло б призвести проєкт у неконсистентний стан.

Багато СКВ забезпечують механізми авторизації та аутентифікації. Адміністратори можуть налаштувати, хто і які операції може виконувати (читання, запис, злиття тощо) [4]. Наприклад, у централізованих системах часто є списки доступу до репозиторіїв, а в Git-платформах (GitHub, GitLab) – можливість налаштувати права на гілки чи запити на внесення змін (pull requests).

СКВ тісно інтегруються з іншими DevOps-інструментами. Зазвичай вони підтримують роботу з системами безперервної інтеграції (CI/CD), IDE та проєктними трекерами. Наприклад, Git можна підключити до Jenkins чи GitLab CI для автоматичних збірок, а сервісні плани (GitHub Issues, Jira) можуть прив’язувати коміти до задач. Такі зв’язки спрощують розгортання й тестування проєкту.

Також СКВ надають можливості налаштовувати робочого процесу команди (GitFlow, trunk-based development тощо). Розподілені СКВ дозволяють

мати кілька віддалених репозиторіїв одночасно, що відкриває можливість організувати розробку за різними моделями ієрархії [4]. Наприклад, команда може використовувати центральний, «головний» сервер для релізів і додаткові особисті форки для експериментів.

Традиційно СКВ ефективніше працюють з текстовим кодом, ніж з великими бінарними файлами. Вставка оновлень великих зображень чи медіа може суттєво збільшити розмір репозиторія. Деякі системи спеціально оптимізовані для бінарних даних (наприклад, Perforce і SVN краще масштабуються для великих бінарних активів [9]), а в Git запропоновано розширення Git LFS для роботи з великими файлами [9].

1.3 Порівняння популярних систем контролю версій

У сучасних фронтенд-проєктах найчастіше використовують Git, Mercurial, SVN і Perforce. Кожна система має свої переваги та недоліки.

СКВ Git характеризується як розподілена, швидка, з потужним гілкуванням і широким набором інструментів. Git став стандартом для веброзробки: до 95% розробників використовують її як основну СКВ [9]. До основних переваг можна віднести підтримку офлайн-операцій (уся історія локально), гнучкі гілки та злиття, велику спільнота та платформи (GitHub, GitLab), інтуїтивні запити на внесення змін у коді (pull requests). У той же час, система має власні недоліки: досить круту криву навчання, складні командні поняття (rebase, merge й ін.), а в «чистому» Git слабша підтримка великих бінарних файлів без додаткових плагінів [9]. У контексті фронтенду Git зручний для керування кодом, HTML/CSS/JS. Проте при частих оновленнях зображень чи дизайнів може розростатися репозиторій – тоді варто застосовувати Git LFS або інші рішення.

Ще однією розподіленою СКВ є Mercurial, проте вона базується на філософії синхронної роботи. Mercurial простіша для освоєння, ніж Git, та має більш зрозумілий інтерфейс команд [9]. Це має свої переваги, якщо в команді є

новачки чи нетехнічні користувачі. Однак Mercurial не набув такої популярності, як Git, його частка на ринку становить близько 2% [9], і навіть Atlassian (Bitbucket) припинила підтримку Mercurial. В умовах фронтенд-розробки Mercurial надає схожі можливості, що й Git (лінійний або гілковий робочий процес), але сьогодні вибір на його користь трапляється рідше. До недоліків також можна віднести меншу кількість хостингових сервісів та поступове згасання підтримки.

На противагу попереднім рішенням, система контролю версій SVN (Apache Subversion) є централізованою системою з дельтовим зберіганням. Перевагами SVN є простота розуміння (у зв'язку з єдиним сховищем), відсутність складних гілок (простий лінійний робочий процес, як у першій лінії розвитку) [10], поширеність у корпоративному середовищі. Спільнота SVN менша, ніж у Git, але багато компаній досі підтримують існуючі SVN-репозиторії. Серед недоліків – необхідність мережевого доступу до сервера для комітів, повільніші операції з історією (коміти й оновлення через мережу), і базова підтримка гілок/злиття робить зливання складнішим [10]. У фронтенд-розробці SVN може бути корисний для невеликих команд із простим робочим процесом; він легко інтегрується з багатьма IDE та інструментами, але поступається в гнучкості Git при роботі з багатьма гілками.

Ще однією альтернативою є Perforce (Helix Core). Це комерційна централізована СКВ, розроблена для великих проєктів з масивними артефактами. Перевагами Perforce є відмінна підтримка великих бінарних файлів і артефактів [9], розумна система блокування файлів, потужний контроль доступу й безпека, а також власна модель гілок (Streams) для зручних злиттів [9]. Серед недоліків можна виокремити високу складність налаштування, комерційну ліцензію та відсутність розподіленості (коміти завжди на сервері). Через ці причини в фронтенд-середовищі Perforce використовують рідше. Зазвичай його обирають лише для надвеликих проєктів з великою кількістю мультимедіа (ігрова індустрія, VFX тощо) [9]. Якщо фронтенд-стек містить багато медіа-файлів і потрібні гарантії цілісності, Perforce може бути

виправданою інвестицією. Нижче в таблиці 1.1 наведено порівняння основних властивостей і відмінностей цих систем.

Таблиця 1.1 – Основні властивості й відмінності розглянутих СКВ

Характеристика	Git	Mercurial	SVN (Subversion)	Perforce
Архітектура	Розподілена (DVCS)	Розподілена (DVCS)	Централізована (CVCS)	Централізована (CVCS)
Модель зберігання	Знімкова (snapshot) [4]	Знімкова (snapshot)	Дельтова (зберігає зміни)	Дельтова (елементи файлів)
Швидкість (локальні оп.)	Дуже висока (усі операції локальні) [4]	Висока (усі операції локальні)	Середня (мережеві коміти)	Висока (оптимізовано під великі дані) [9]
Офлайн-робота	Підтримується повністю	Підтримується повністю	Обмежена (неможливо коміт без зв’язку)	Немає (завжди потрібен сервер)
Гілкування / злиття	Дуже зручне, гнучке[9]	Підтримується, але простіше	Прості гілки, менше гнучкості [10]	Підтримується (Streams)
Легкість використання	Складна, крута крива навчання [9]	Простий інтерфейс, легше для новачків [9]	Простий для базових користувачів	Складне, більше для досвідчених
Масштабування	Добре масштабується (відкритий код, розгалуженість)	Добре масштабується (подібно до Git)	Обмежене (найкраще для середніх проєктів)	Високоефективне (призначене для великих проєктів) [9]
Підтримка бінарних	Слабка без розширень (але є Git LFS) [9]	Середня	Добра (підтримує блокування файлів) [9]	Відмінна (оптимізовано під великі бінарні активи) [9]
Приклад використання	Web/мобільний софт, відкритий код[9]	Альтернатива Git (де потрібно простіше)	Корпоративні проєкти, де потрібен простий контроль	Ігрова індустрія, VFX, де є великі активи
Сервіси та інструменти	GitHub, GitLab, Bitbucket; інтеграція з CI/CD[9]	Bitbucket (обмежено), деякі корпоративні	Subversion сервера (Assembla, Apache SVN)	Helix Core server; інтеграція з IDE (P4VS тощо)

Загалом, у контексті фронтенд-розробки Git (у поєднанні з хостингом на GitHub/GitLab) є безперечним лідером: він пропонує високу швидкість, розвинені можливості для колективної роботи та інтеграцію з сучасними CI/CD інструментами [9]. Mercurial може стати легшим варіантом для команд-початківців, але в Україні та світі зараз перевага віддається саме Git. SVN ще зустрічається в організаціях зі стабільними процесами і простим workflow, але поступово відходить у бік DVCS. Perforce вартий уваги, якщо багато графічних чи медіа-файлів: він спеціально створений для цього, але його застосування рідше виправдане для звичайних фронтенд-проєктів через високу складність і вартість [9]. Тому вибір СКВ залежить від специфіки проєкту. Для більшості сучасних фронтенд-команд оптимальним є Git (розподілена, потужна система) з увагою до оптимізації великих файлів. Між тим, знання відмінностей архітектур і моделей зберігання допомагає застосувати правильний інструментарій під конкретні задачі.

РОЗДІЛ 2
ОСОБЛИВОСТІ ПРОЦЕСУ РОЗРОБЛЕННЯ ФРОНТЕНДУ

2.1 Специфіка розроблення фронтенду

Розроблення фронтенду традиційно передбачає модульність і багатофайловість: код розділений на численні компоненти та модулі (наприклад, React-компоненти, Angular-модулі), що дозволяє тримати кожну частину логіки у власному файлі. Завдяки цьому проєкт із тисячами рядків коду може розбиватися на дрібніші файли, полегшуючи навігацію та спільну роботу. У такому підході важливу роль відіграє використання пакетних менеджерів (наприклад, npm), які зберігають інформацію про залежності у файлі package.json. Пакетний менеджер автоматично завантажує необхідні модулі та фіксує їх версії у файлах блокування (наприклад, package-lock.json), що забезпечує детерміноване збирання та усуває невідповідності середовищ розробки.

Фронтенд-розробка відзначається динамічністю змін: інтерфейс користувача регулярно коригується й покращується у процесі розробки, часто застосовуються «гарячі» перезавантаження (Hot Module Replacement, HMR) для швидкого відображення змін без повного перезапуску сторінки. Наприклад, Webpack DevServer із включеним HMR дозволяє автоматично оновлювати частини додатку в браузері при зміні коду, зберігаючи стан додатку. Такий підхід пришвидшує цикл розробки й тестування, але потребує налаштування інструментів, які слід підтримувати актуальними.

Сучасний фронтенд тісно пов’язаний з набором спеціалізованих інструментів. У межах JavaScript-екосистеми зазвичай використовують базовий менеджер пакетів npm (Node Package Manager), який надає доступ до величезної кількості бібліотек і фреймворків (React, Angular, Vue тощо). Через npm встановлюються і власне фреймворки, й утиліти для збирання проєкту (Webpack,

Babel, Gulp) та пакети для підтримки якості коду (ESLint, Jest, Prettier). Типова взаємодія розробників на основі JSON-конфігурацій показана на рис. 2.1.

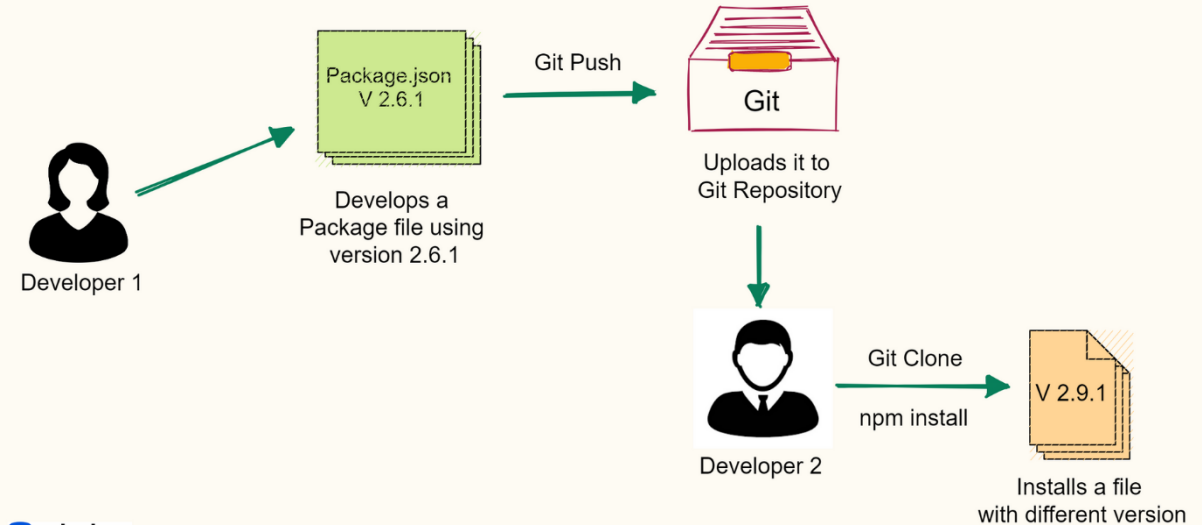


Рисунок 2.1 – Взаємодія розробників фронтенду з урахуванням JSON-конфігурацій [11]

Webpack – модульний бандлер, який збирає окремі файли (модулі) в один або декілька фінальних бандлів для браузера. Він аналізує залежності (імпорти) в коді, створює граф залежностей та об’єднує ресурси (JS, CSS, зображення) у оптимізовані пакети. Наприклад, у проєктах на React чи Vue Webpack часто налаштовується через конфігурацію webpack.config.js, де вказуються точки входу, завантажувачі (loaders) для трансформації файлів та плагіни для оптимізації.

На рис. 2.2 показано сучасний підхід побудови фронтенду на основі мікрофронтендів. Ця стратегія передбачає, що монолітний додаток розбивається на незалежні модулі, які розробляються, тестуються і розгортаються окремими командами. За аналогією з мікросервісами, кожен модуль відповідає за певний бізнес-домен, але разом формує єдиний інтерфейс. Модулі об’єднуються на етапі компіляції, після чого виходить один збірний артефакт.

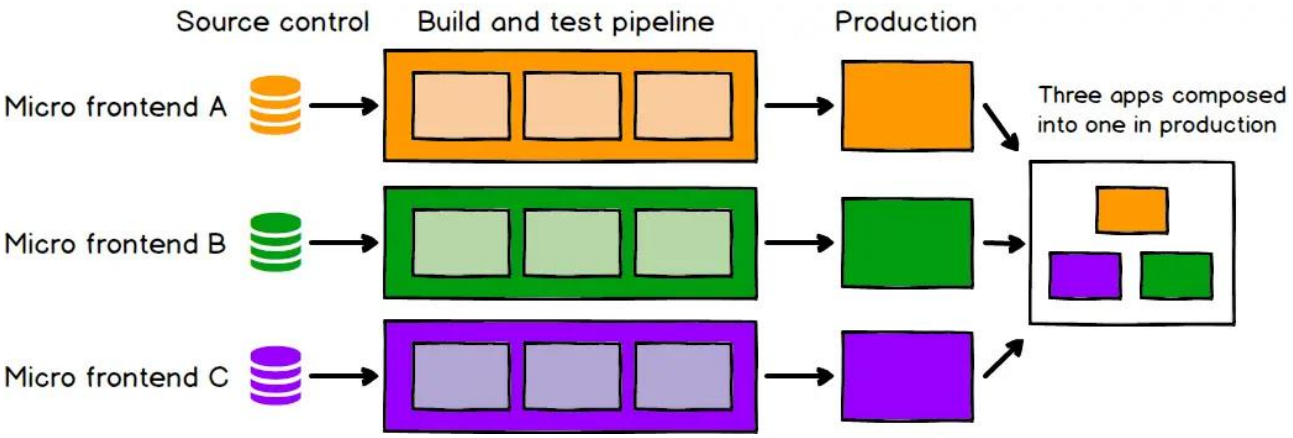


Рисунок 2.2 – Типовий конвеєр збирання фронтенду [12]

Іншим важливим компонентом розроблення фронтенду є Babel – компілятор JavaScript, що дає змогу використовувати найсучасніші можливості мови, перетворюючи їх на код, сумісний із старішими браузерами. У поєднанні з Webpack для обробки JS-файлів встановлюється babel-loader та відповідні пресети (наприклад, @babel/preset-env), що дозволяє писати код на останньому стандарті ECMAScript, а Babel забезпечує зворотну сумісність.

Прикладом є конфігурації багатьох сучасних CLI. Середовище Create React App за замовчуванням використовує Babel і Webpack, а Vue CLI та Angular CLI (побудований над Webpack) також інтегрують ці інструменти у свої шаблони. Angular, до того ж, офіційно базується на TypeScript – надбудові над JavaScript зі статичною типізацією. TypeScript дозволяє описувати типи даних у коді, що підвищує надійність та зрозумілість проєктів. Фреймворки Angular і Vue (з версії 3) широко підтримують TypeScript; у React-проєктах також часто додають TypeScript за допомогою @types/react і подібних пакетів. Наприклад, TypeScript-версія компонента в React може виглядати так, як у лістингу 2.1.

Лістинг 2.1 – Зразок коду TypeScript-версії React-компонента

```
tsx
function MyButton({ title }: { title: string }) {
  return <button>{title}</button>;
}
```

Статичний аналізатор коду ESLint перевіряє JavaScript/TypeScript на помилки та невідповідність стилю. Він дозволяє команді визначати і дотримуватися єдиних стандартів кодування: виявляє синтаксичні помилки, непотрібні змінні, питання стилю (двокрапку, пробіли тощо) та інші «підозрілі» конструкції. ESLint особливо корисний у великих проєктах: він автоматизує раннє виявлення проблем, забезпечує однаковість стилю в команді та підвищує читабельність коду. Наприклад, у React-проєктах часто використовують плагіни eslint-plugin-react та plugin:@typescript-eslint/recommended для підтримки специфічних правил React і TypeScript.

Загалом, кожен із цих інструментів зв’язаний із фреймворками: зокрема, React має вбудовану інтеграцію з Babel/TypeScript (через create-react-app або інші шаблони), Angular відпочатку створений з TypeScript і використовує Webpack, а Vue надає CLI, який «під капотом» працює з Webpack і Babel. Звідси, сучасний фронтенд-процес характеризується використанням npm-пакетів, транспіляції коду (Babel/TypeScript), пакування ресурсів (Webpack) і перевірки якості (ESLint).

2.2 Проблеми в командній роботі

При розробці фронтенду додатків командно виникають кілька типових проблем. Першою з них є конфлікти при злитті**.** При великій кількості розробників, які паралельно вносять зміни в кодову базу, часто з’являються конфлікти Git (merge conflicts). Особливо це стосується довготривалих гілок: якщо кілька контрибуторів працюють зі спільним файлом, після злиття з гілкою main потрібне додаткове вирішення конфлікту. Щоб зменшити конфлікти, рекомендують робити часті «pull» та «merge» або обмежувати час життя гілок. Інструменти, як Git або мережеві платформи (GitHub/GitLab), допомагають ідентифікувати й вирішувати такі проблеми, але це все одно додатково витрачає час.

Як уже було зазначено, суттєву роль відіграє синхронізація залежностей. Потрібно гарантувати, що всі члени команди використовують однакові версії бібліотек. Різні версії Node.js чи пакетів можуть призвести до того, що переносимість проєкту страждатиме. Уникнути цього допомагають lock-файли (package-lock.json, yarn.lock): вони фіксують точну версію кожної залежності та її ієрархію. Наприклад, усім розробникам потрібно регулярно оновлювати локальний репозиторій (git pull) і перевстановлювати пакети (npm ci), щоб стежити за змінами в package.json. У реальних проєктах часто використовують автоматизованих ботів (напр. Dependabot, Renovate) для оновлення залежностей без конфліктів, що мінімізує «dependency hell».

Окрему увагу варто приділити артефактам збирання. Вихідні (збіркові) файли, наприклад, результати роботи Webpack (bundle.js, мініфіковані скрипти тощо) зазвичай не зберігаються в репозиторії. Проте помилки в налаштуванні .gitignore можуть призводити до їх випадкового коміту. Це збільшує розмір репозиторію й може ускладнювати оновлення коду. Також важливо налаштувати CI/CD-конвеєри (Jenkins, GitHub Actions, GitLab CI тощо), щоб під час «збірки» автоматично видаляти застарілі артефакти (clean build) і генерувати свіжі.

Також слід враховувати, що різні розробники мають різні звички в написанні коду, тому без єдиних правил кодова база стає «пухкою». .частково проблему вирішують стиль-гайди (Airbnb, Google JS style) та засоби автоматизації: ESLint/Prettier (або як згадано, ESLint) встановлюють «єдине джерело істини» для стилю коду. Після введення спільної ESLint-конфігурації з автоматичним виправленням (autofix) подібні суперечки зникають.

У великих проєктах паралельно можуть працювати кілька команд над різними модулями (наприклад, одна пише UI, інша – окремі віджети). Необхідне чітке планування (задачі в Jira/GitHub Issues), регулярні синхронізації (stand-up, спринт-планування) та розподіл відповідальності (code owner-и за частини системи). Часто застосовують гнучкі методи (Scrum/Kanban) для координування, а також скрипти чи «watch»-інструменти Webpack для автоматичного оновлення додатку при змінах, що допомагає уникнути «зрізання кроків» у тестуванні.

Процес розгортання теж потрібно розглядати як окрему проблематику. Фронтенд-додатки розгортаються в декілька середовищ (dev, staging, production). Важливо синхронізувати версії: наприклад, якщо бекенд API змінився, фронтенд повинен «розуміти» оновлені версії. Проблеми з розгортанням можуть бути пов’язані з некоректними конфігураціями Webpack (відсутність мінімізації, неправильні шляхи до ресурсів) або з кешуванням у браузері. Щоб мінімізувати ризики, часто використовують автоматичні CI/CD-процеси, де фронтенд збирається і розгортається на тестовий сервер після кожного злиття в головну гілку.

Як приклад реального проєкту, можна розглянути роботу над інтернет-магазином на React. Команда розробників стикалася з проблемою невідповідності версій пакетів: на одній машині встановлено React 17, на іншій – React 18. У результаті виникають помилки в браузері й невдачі побудови. Ситуацію можна вирішити фіксацією версій у package.json та оновленням package-lock.json командою, а також викликом команди npm ci, що забезпечує відтворювану збірку.

Інший випадок – різні стандартні правила: частина розробників віддавали перевагу крапці з комою в кінці виразу, інша – ні. Це створювало конфлікти на етапі рев’ю. Після інтеграції Prettier/ESLint з однією версією конфігураційних файлів і включення перевірок в ході неперервної інтеграції, дискусії про форматування відпадають, а якість коду залишилася високою. Такі підходи (суворий контроль версій залежностей та єдині правила коду) в реальних командах мінімізують «внутрішні» розбіжності і пришвидшують роботу.

2.3 Підходи до управління версіями

Для організації спільної роботи над кодом використовують різні моделі гілкування та структури репозиторіїв. Найпростішою організаційно є централізоване сховище в SVN-подібному стилі. У такому разі існує лише одна головна гілка (наприклад, main або master), і всі розробники фіксують зміни в

неї. Така схема схожа на роботу в SVN: відсутні допоміжні гілки, код одразу потрапляє в центральний репозиторій. Перевагами є простота в освоєнні, що має сенс для маленьких команд і швидкого старту. Кожен розробник отримує повну локальну копію проєкту і може самостійно фіксувати зміни, не залежачи від інших. Недоліки полягають у тому, що головна гілка завжди може бути нестабільною (відсутність перевірок перед злиттям), високий ризик конфліктів при паралельній роботі. Такий підхід не забезпечує ізоляції для нових функціональних можливостей чи виправлень помилок.

Більш упорядкованим є підхід з використанням функціональних гілок (feature branch). За такої моделі для розроблення кожної нової функції чи виправлення створюється окрема гілка (feature/назва, bugfix/назва тощо). Розробник не комітить безпосередньо в main-гілку, а відгалужується від неї і після завершення зливає свою гілку через pull request або merge. Це дозволяє групам розробляти нововведення незалежно, головна гілка не містить «зламаного» коду, що полегшує безперервну інтеграцію та тестування. Код з функціональних гілок інтегрується до основної гілки повільно, оскільки для цього необхідне його попереднє рецензування іншим членом команди. Також потрібна дисципліна: якщо гілка живе довго і синхронізується рідко, можуть виникнути відставання і складні конфлікти.

Більш розширеною моделлю галуження є Gitflow, що вводить спеціальні гілки для основних стадій релізу (зазвичай master, develop, release/*, hotfix/*, feature/*) [15]. Зазвичай гілка develop слугує інтеграційною гілкою, а головна гілка зберігає тільки виробничі релізи (з тегами версій). Загальна схема галуження відображена на рис. 2.3.

Чітка структура гілок і процесу релізу зручна для великих проєктів з регулярними випусками. Окремі release-гілки дозволяють готувати продакшн-версії без переривання розробки нових фіч. Завдяки цьому знижується ризик введення помилок у стабільну версію [15]. Проте подібна стратегія зараз втрачає популярність через досить високу складність та громіздкість моделі. Багато гілок довготривалого життя спричиняє суттєву складність у керуванні. Це може

уповільнювати розробку через необхідність ручного управління гілками і частих зливань. У ситуаціях, коли потрібні швидкі і часті релізи (наприклад, CI/CD і «деплой кожну зміну»), Gitflow може бути надто формальним. Порівняння команд для Git Flow та традиційного Git наведено в таблиці 2.1.

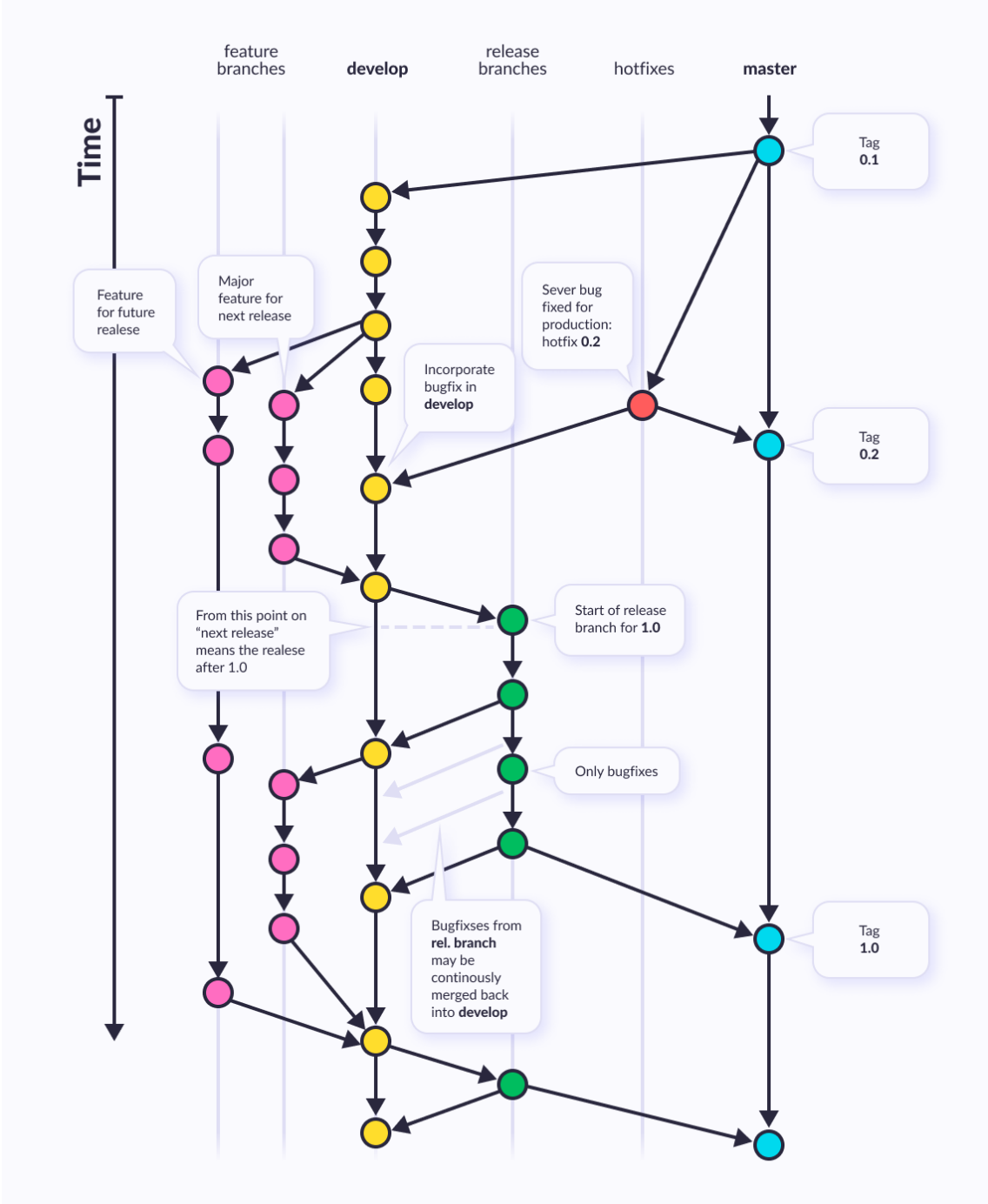


Рисунок 2.3 – Схема галуження стратегії Git Flow [13]

Таблиця 2.1 – Порівняння команд Git Flow та традиційного Git

Дія	Git flow	Git

Ініціалізація репозиторію	`git flow init`	`git init`
		`git commit --allow-empty -m "Initial commit"`
		`git checkout -b develop master`

Підключення до віддаленого репозиторію	-	`git remote add origin git@github.com:MYACCOUNT/MYREPO`
Створення функціональної гілки (feature branch)	`git flow feature start MYFEATURE`	`git checkout -b feature/MYFEATURE develop`
Публікація (Share) функціональної гілки	`git flow feature publish MYFEATURE`	`git checkout feature/MYFEATURE` `git push origin feature/MYFEATURE`
Отримання актуальної (latest) функціональної гілки	`git flow feature pull origin MYFEATURE`	`git checkout feature/MYFEATURE` `git pull --rebase origin feature/MYFEATURE`

Фіналізація функціональної гілки	`git flow feature finish MYFEATURE`	`git checkout develop`
		`git merge --no-ff feature/MYFEATURE`
		`git branch -d feature/MYFEATURE`

Пуш злитої функціональної гілки	-	`git push origin develop` `git push origin :feature/MYFEATURE (якщо відбувся пуш)`
Створення гілку випуску (release)	`git flow release start 1.2.0`	`git checkout -b release/1.2.0 develop`
Розшарювання гілки випуску	`git flow release publish 1.2.0`	`git checkout release/1.2.0` `git push origin release/1.2.0`
Отримання найбільш свіжої версії гілки випуску	-	`git checkout release/1.2.0` `git pull --rebase origin release/1.2.0`

Фуналізація гілки випуску	`git flow release finish 1.2.0`	`git checkout master`
		`git merge --no-ff release/1.2.0`
		`git tag -a 1.2.0`
		`git checkout develop`
		`git merge --no-ff release/1.2.0`
		`git branch -d release/1.2.0`

Пуш злитої гілки випуску	-	`git push origin master`
		`git push origin develop`
		`git push origin --tags`
		`git push origin :release/1.2.0 (якщо відбувся пуш)`

Створення гілки виправлень (hotfix)	`git flow hotfix start 1.2.1 [commit]`	`git checkout -b hotfix/1.2.1 [commit]`

Фіналізація гілки виправлень	`git flow hotfix finish 1.2.1`	`git checkout master`
		`git merge --no-ff hotfix/1.2.1`
		`git tag -a 1.2.1`
		`git checkout develop`
		`git merge --no-ff hotfix/1.2.1`
		`git branch -d hotfix/1.2.1`

Пуш злитої гілки виправлень	-	`git push origin master`
		`git push origin develop`
		`git push origin --tags`
		`git push origin :hotfix/1.2.1 (якщо відбувся пуш)`

Популярність Gitflow стала знижуватися під впливом магістральних робочих процесів, які сьогодні вважаються кращими для сучасних схем неперервної розробки ПЗ і застосування DevOps. У Git-flow використовуються довготривалі функціональні гілки та кілька основних гілок. Загалом використовується більше гілок, їх термін життя довше, а коміти в них більші, ніж у моделі магістральної розробки. Відповідно до цієї моделі, розробники створюють функціональну гілку і відкладають її злиття з головною магістральною гілкою до завершення роботи над функцією. Такі довгострокові функціональні гілки вимагають більш тісної взаємодії розробників при злитті, оскільки створюють підвищений ризик відхилення від магістральної гілки та виконання конфліктуючих оновлень.

Модель Gitflow також передбачає окремі напрямки основних гілок для розробки, термінових виправлень, функцій та релізів. Для злиття комітів між цими гілками застосовуються різні стратегії. Оскільки більша кількість гілок створює складнощі при взаємодії та управлінні, у таких системах часто виникає потреба у додаткових нарадах щодо планування та перевірок з боку команди.

З розвитком вебдодатків та неперервного постачання на себе звернули увагу легші та швидші моделі галуження, зокрема GitHub Flow. Вона концентрується навколо основної вітки та функціональних гілок. Main завжди перебуває в готовому до випуску стані. Розробники створюють нову функціональну гілку для внесення змін. Після огляду цих змін на дефекти та якість коду, вони будуть спрямовані (deploy) в продакшн. Якщо дані зміни створюють проблеми, відбувається відкочування до поточної версії коду (гілки main). Інакше, функціональна гілка буде злита в гілку main (рис. 2.4).

У порівнянні з Git Flow, GitHub Flow набагато менш важковаговий. Супровід відносно простий, оскільки єдиною довготривалою гілкою є main. Проте GitHub Flow успадковує інші недоліки Git Flow. Недисципліновані команди, які тримають функціональні гілки відкритими тижнями, ризикують мати серйозні проблеми при злитті гілок. Загалом, неуспішне злиття може

залишити основну гілку в непрацездатному (undeployable) стані, що є неприпустимо при неперервному постачанні.

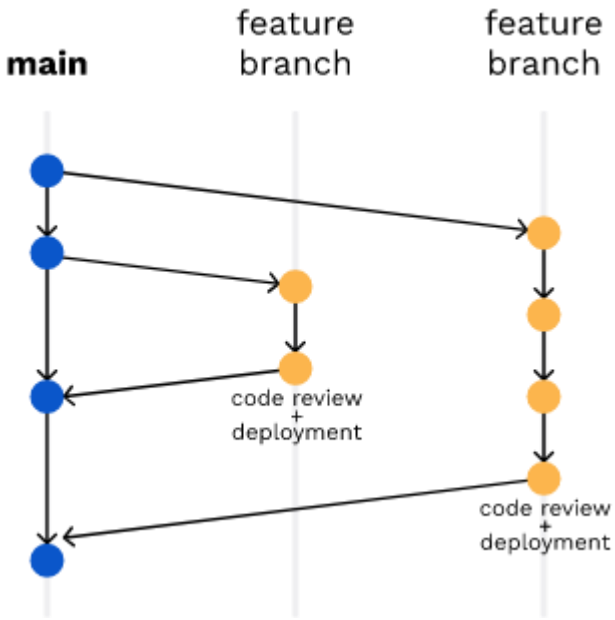


Рисунок 2.4 – Модель галуження GitHub Flow

Отже, щодо стратегії галуження GitHub Flow можна сказати, що вміст гілки master завжди працездатний (deployable). Починаючи роботу над чимось новим, слід відгалужувати від гілки main нову гілку, назва якої буде відповідати її призначенню. Зафіксувавши цю гілку локально, варто відправляти свою роботу регулярно в одноіменну гілку на сервері. При потребі в відгуку чи допомозі, або коли розробник вважатимете гілку готовою до злиття, він відправляє запит на злиття. Після того, як хтось із команди переглянув та погодив підготовану функціональну можливість, розробник може злити свою гілку в main. У разі успіху нова функціональність може негайно бути впровадженою в продакшн.

Поширеною моделлю в нинішніх умовах також є магістральна розробка (Trunk-Based Development), де всі розробники працюють переважно з однією гілкою (master/main) і зливають туди невеликі, часті оновлення. Іноді допускаються дуже короткоживучі функціональні гілки для кількох дрібних комітів, які одразу зливають у main після проходження тестів [16]. Ці особливості схематично відображено на рис. 2.5.

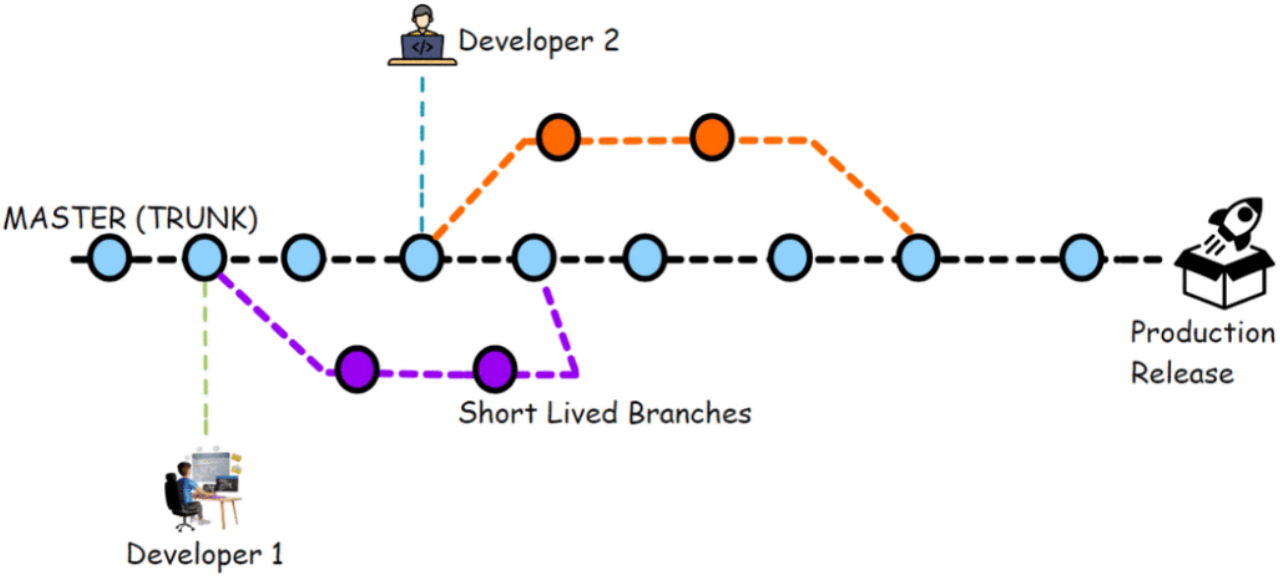


Рисунок 2.5 – Схема галуження магістральної розробки [14]

Цей підхід дозволяє команді швидко бачити загальні зміни та уникати ситуацій, коли гілки довго існують окремо й потім важко об’єднуються. Головна гілка завжди залишається актуальною і готовою до розгортання, що особливо зручно для CI/CD процесів. Щоб нові функціональні можливості не ламали основну версію, навіть якщо вони ще не завершені, розробники використовують функціональні перемикачі (feature flags). Це частини коду, які дозволяють вмикати або вимикати певну функціональність без зміни основного коду (рис. 2.6). Наприклад, така функціональність може бути задеплоєна, але доступна тільки для тестувальників або обмеженої групи користувачів. Цей підхід особливо зручний при роботі над MVP, стартапами або в умовах, коли проєкт постійно змінюється. Але потрібно, щоб у команді був порядок, і всі чітко дотримувались правил. Інакше помилка одного розробника може зіпсувати роботу всім.

На практиці функціональний перемикач зазвичай є булевою змінною або парою «ключ-значення» в конфігурації програми. Складніші варіанти включають налаштування вибору аудиторії, поступове розгортання тощо. Типологія функціональних перемикачів представлена в [30]:

– перемикачі випуску функцій (release feature flags) використовуються для контролю розгортання нових функцій; прапорець часто знімається після того, як функція повністю розгорнута та визнана стабільною;

– перемикачі операційних функцій (operational feature flags) застосовуються для керування операційними аспектами системи, такими як активація резервного механізму в разі системного збою;

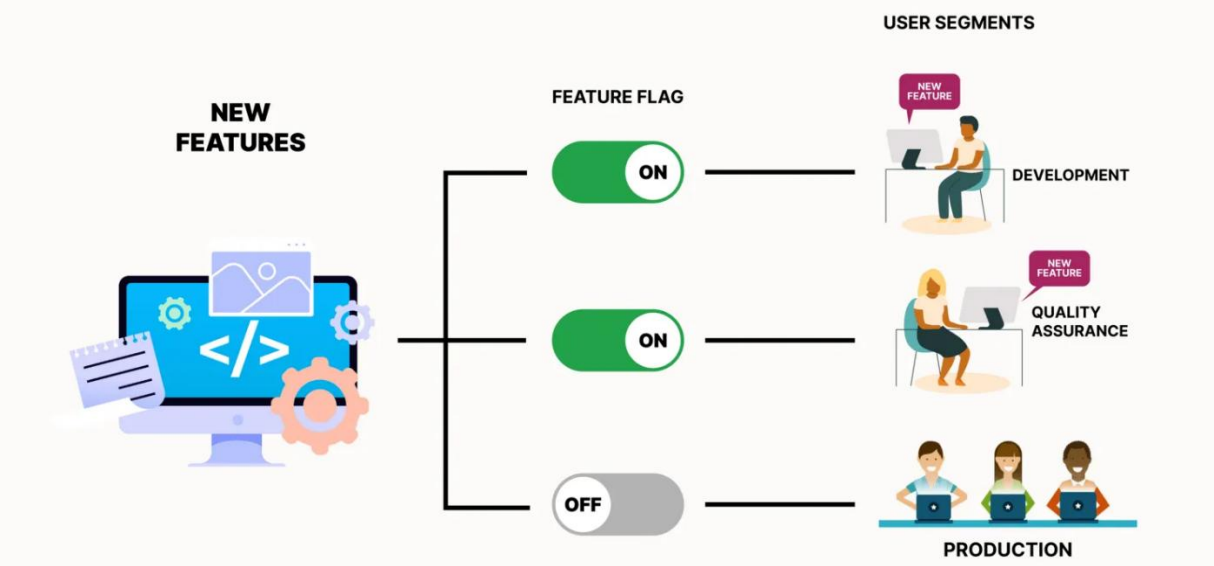


Рисунок 2.6 – Принцип роботи функціональних перемикачів

– перемикачі експериментальних функцій (experimental feature flags) задіюються для A/B-тестування або канарейкового тестування, де різні варіанти функцій порівнюються одна з одною за ефективністю; вони використовуються для перевірки гіпотези нової функції перед її розгортанням для всіх користувачів;

– перемикачі дозволів на доступ до функцій (permission feature flags): контролюють доступ до функцій на основі ролі або дозволів користувача (наприклад, безкоштовний та преміум-рівні); коли користувач намагається отримати доступ до преміум-функції, система спочатку перевіряє стан прапорця функції «преміум-доступ» і не дозволить доступ, якщо прапорець має значення «false».

Результати аналізу стратегій галуження узагальнено в порівняльній таблиці 2.2.

Таблиця 2.2 – Порівняльний аналіз моделей галуження

Модель	Опис	Переваги	Недоліки
Централізована	Одна головна гілка (main), всі коміти сюди (аналог SVN).	Дуже проста, не вимагає складних процедур, підходить малим командам.	Головна може бути нестабільною, високий ризик конфліктів, відсутня ізоляція функціональних можливостей.
Функціональні гілки	Окремі гілки для кожної фічі/багфіксу, злиття через pull-request.	Розробка ізольована від main, гарантує стабільність головної гілки, зручна для рев'ю.	Вимагає частого зливання та оновлення гілок; довгі фічі можуть спричиняти конфлікти.
Gitflow	Строге розгалуження: є develop, master, feature/*, release/*, hotfix/*[15]	Чітка структура релізів і версіонування; підходить для проєктів з регулярними релізами.	Складність і громіздкість процесу; багато гілок, повільніший розвиток через бюрократію.
Магістральна (Trunk-Based) розробка	Всі розробники мерджать невеликі зміни безпосередньо до main.	Швидка інтеграція і доставка, проста модель, сприяє частій поставці оновлень.	Ризик одразу «ламати» головну гілку; потребує високої дисципліни розробників.

У цьому розділі було детально проаналізовано особливості процесу розробки фронтенду, зокрема питання модульності, багатофайловості та організації коду. Встановлено, що для ефективної розробки доцільно використовувати компонентний підхід, який забезпечують такі сучасні фреймворки, як React. Обрано саме React через його популярність, зручність у роботі з компонентами та велику спільноту розробників, що полегшує підтримку проєкту.

Також було обґрунтовано вибір системи контролю версій Git, яка забезпечує надійне керування змінами, дозволяє працювати команді розробників узгоджено та безпечно, а також спрощує процес інтеграції коду. Запропоновано подальше впровадження GitFlow та магістральної розробки як зручних схем

роботи з гілками, які дозволяють ефективно організувати розробку, тестування та випуск програмного забезпечення.

Отже, врахування специфіки фронтенд-розробки, а також правильний вибір технологій і методологій управління кодом, є важливими складовими успішної реалізації програмного продукту, що забезпечує якість, масштабованість і стабільність роботи.

РОЗДІЛ 3
ВПРОВАДЖЕННЯ СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ У РОЗРОБЛЕННЯ ФРОНТЕНДУ

3.1 Методологія впровадження системи контролю версій

Впровадження СКВ слід проводити за чітким планом. Нижче наведені основні етапи:

1. Аналіз потреб. Оцінюють поточні процеси і вимоги: розміри команди, географічна децентралізація, кількість проектів, необхідність роботи з великими файлами тощо. Аналіз включає вивчення витрат на ліцензії та підтримку, наявність тренінгів для команди, рівень автоматизації процесів і практик CI/CD [18]. Слід обговорити, чи потрібно блокування файлів (для великих бінарних файлів), хто виконуватиме адміністрування СКВ, чи існує потреба у збереженні історії з старої системи.

2. Підготовка інфраструктури. Визначаються сервер або хмарна платформа для зберігання репозиторіїв (GitLab/GitHub/Bitbucket чи власний сервер). Налаштовують права доступу, SSL-сертифікати, мережеві політики та резервне копіювання. Для великих команд часто організовують mirror-сервери (Git Smart Mirroring) для швидкого доступу з різних регіонів. Враховують масштабованість: наприклад, для Perforce рекомендується окремий high-performance сервер під великий датасорс. Також планують середовища CI/CD (виділені вузли для побудов і тестів).

3. Структура репозиторію. Формують стандартну директорію проєкту. У випадку фронтенду зазвичай створюють src/ для коду, assets/ для зображень/стилів, docs/ для документації та конфігураційні файли (наприклад, package.json з npm-залежностями) [19]. Встановлюють правила .gitignore (ігнорування node_modules, збірок, тимчасових файлів) [17], щоб не зберігати зайвий вміст у СКВ. Чітка ієрархія директорій (приклад: розділення компонентів, утиліт, сторінок) допомагає уникнути хаосу і полегшує навігацію в коді[19].

4. Навчання команди. Всіх розробників слід ознайомити з обраною СКВ: базовими командами (clone, commit, push, pull/fetch), моделлю гілок, процесом рев'ю та політиками злиття. Проводять семінари або воркшопи з демонстрацією робочого процесу. Полезно розробити внутрішню документацію з правилами оформлення комітів, іменування гілок та використання CI/CD. Наприклад, в DreamHost описуються критичні Git-команди українською, що допомагає початківцям швидше втягнутися[20]. Команда також навчається користуватися Git-клієнтами або IDE, де інтегрована підтримка СКВ (наприклад, Sourcetree, GitKraken, VS Code).

5. Міграція. Якщо код вже був у старій СКВ, його переносять у нову. Рекомендується планувати перенесення на час, коли розробка призупинена (наприклад, вихідні)[18]. Перед міграцією створюють бекап старого репозиторію та проганяють тестове імпортне. При перенесенні можуть застосовуватись спеціальні утиліти для експорту історії (для Git є [git svn], git p4, інші скрипти). Після імпорту перевіряють цілісність даних: чи всі коміти і гілки присутні, чи коректно працює збірка коду, чи збережені посилання між файлами. DevOps-керівники радять зберігати доступ до старої системи принаймні протягом деякого часу, щоб у разі потреби мати можливість звернутися до старої історії [18]. Також документують процес міграції (лог файлів, примітки до ребейзів) у README нового репозиторію.

6. Повне впровадження. Після успішної міграції і верифікації розгортання, переводять усю розробку на нову СКВ. Блокують роботу в старій системі, призначають відповідальних за мерджі. Встановлюють додаткові правила: наприклад, захист головної гілки (main), вимогу підтвердження CI-проходів перед злиттям, автоматичні перевірки pull request. Налаштовують сповіщення (анонси у чатах, email) про стан збірок та тестів.

7. Оптимізація. Після впровадження важливо збирати зворотний зв'язок і вдосконалювати процес. Наприклад, стежать за часом виконання git-операцій і розміром репозиторію (за потреби підключають Git LFS або очищають історію). Регулярно оновлюють правила .gitignore під нові файли, додають pre-

commit-хуки для лінтингу та тестів. Можуть впроваджувати інструменти автоматизації (Commitizen, commitlint) для контролю формату комітів, якість коду. Також варто періодично оновлювати навчальні матеріали та контролювати дотримання стандартів усією командою.

Як уже було зазначено, для впорядкування розробки зазвичай визначають єдину основну гілку (звичайно main). Далі використовують гілки типу feature/* для розробки нових фіч, release/* для стабілізації випуску та hotfix/* для термінових виправлень. Наприклад, у моделі Git Flow кожен feature-блок створюється від develop (або main), у ньому роблять зміни, а після перевірки роблять pull/merge request для злиття назад. Можливі стратегії гілок різного життєвого циклу (довготривалі й короткотривалі feature-branches, окрема гілка розробки develop) [18]. Важливо дотримуватися іменування: наприклад, feature/auth-login або bugfix/navbar-display. Гілки завжди мають відображати зміст змін або номер завдання для прозорості. Приступаючи до роботи, розробник робить git pull для актуалізації локальної копії, створює гілку, робить коміти, потім зливає гілку через Pull Request. Перед злиттям бажано оновити гілку з урахуванням змін в main (через merge або rebase) для уникнення конфліктів.

Рекомендують формувати маленькі атомарні коміти – один коміт = одна логічна зміна (наприклад, реалізація однієї задачі або виправлення конкретної помилки). Повідомлення коміту повинно чітко описувати зміни. Наприклад, специфікація Conventional Commits пропонує починати рядок з типу (feat:, fix:, docs: тощо), далі вказувати короткий опис і номер задачі [21]. Це спрощує читання історії і автоматичний аналіз (семантичне версіонування). У повідомленні часто вказують посилання на трекер (напр., PROJ-113), щоб зрозуміти контекст задачі. Слід уникати розпливчастих фраз на кшталт «виправлено» без деталей. Процес коміту повинен проводитися лише після локального запуску тестів та перевірок, тому використовують pre-commit хуки.

Перегляд коду здійснюють через Pull Request, PR (GitHub) або Merge Request (GitLab). Після створення PR необхідно призначити одного-двох

рецензентів. Як радить документація GitLab, всі merge request проходять рев'ю для забезпечення ефективності та безпечності коду [22]. Рецензенти перевіряють стиль і архітектуру змін, додають коментарі та зауваження. Тільки після схвалення одним або кількома рев'юерами і проходження CI-перевірок гілку зливають у цільову. Зазвичай встановлюють правило, що щонайменше один розробник із досвідом чи доменною експертизою має затвердити PR [22]. Коментарі під час рев'ю мають бути конструктивними: підкреслювати як сильні сторони, так і області для поліпшення коду. Після злиття розробник мержа може видалити локальну гілку.

Для автоматичної перевірки коду перед комітами використовують Git hooks – скрипти, які виконуються на певних етапах (pre-commit, commit-msg, pre-push тощо). Практично всі проєкти на Node.js застосовують Husky – npm-бібліотеку для зручного управління хуками. Husky перехоплює команди Git і виконує задані скрипти перед commit або push [15] Наприклад, pre-commit хук може запускати лінтер (ESLint, Prettier) та модульні тести. Для скорочення роботи з перевірками на етапі коміту використовують lint-staged – він запускає ESLint/Prettier тільки на тих файлах, які додані до staging area, і блокує коміт, якщо перевірка не проходить [15]. Отже, Husky + lint-staged автоматизують дотримання правил коду перед кожним commit, знижуючи кількість проблем у головній гілці.

3.2 Інтеграція з інструментами

Системи безперервної інтеграції та доставки (CI/CD) автоматизують збірку, тестування і розгортання коду. Для фронтенду популярні GitHub Actions (вбудований у GitHub), GitLab CI/CD та Jenkins. У GitHub Actions описують workflow у файлах .github/workflows/*.yml, у GitLab – у .gitlab-ci.yml. CI-системи для Node-проєктів зазвичай виконують npm ci (швидке встановлення залежностей), потім npm test і npm run lint на кожному пулл-реквесті або push. Наприклад, в Jenkins є плагіни для Git, які дозволяють запускати збірки і тести

при кожному оновленні репозиторію. GitLab CI має вбудовану підтримку автоматизації, тому проєкт може бути зібраний на власних раннерах без додаткового сервера [10]. Інші хмарні CI (CircleCI, Travis CI) також підтримують Git та npm-сценарії. Внаслідок інтеграції з CI/CD розробники отримують швидкий зворотний зв'язок: зміни перевіряються тестами автоматично при кожному коміті, що знижує ризик помилок у продакшені [19].

У контексті менеджерів пакетів для фронтенду стандартним є npm (або альтернативи Yarn, pnpm). Вони не є частиною СКВ, але з ними тісно інтегровані: перелік залежностей зберігають у package.json, а встановлені пакети ігнорують у .gitignore [17]. У CI-сценаріях виконують npm install (або npm ci) для встановлення залежностей і npm run build для збірки фронтенду. Існують пакети для лінтингу (ESLint, Stylelint), форматування (Prettier) та тестування (Jest, Cypress), які налаштовують через npm-скрипти. Крім того, можна використовувати інтеграції типу Dependabot (GitHub) чи Renovate для автоматичних оновлень залежностей.

Враховуючи потреби в автоматизації тестування, фронтенд-проєкт передбачає модульні й e2e-тести. Як правило, в пайплайнах CI/CD запускають модульні тести (npm test з Jest/Mocha) та UI-тести (Cypress, Selenium). Також запускають перевірки коду на стилі (ESLint). Рекомендується негайно тестувати будь-яку зміну: інструменти CI підхоплюють push і запускають build/test по скриптах з package.json. Така практика дозволяє виявити помилки ще на ранніх стадіях розробки [19].

Для автоматичного випуску нових версій часто застосовують semantic-release. Цей інструмент аналізує повідомлення комітів за Conventional Commits і автоматично обчислює наступний версіонний номер, генерує changelog та публікує пакет (наприклад, на npm) [23] Semantic-release слідує принципам Semantic Versioning, тож, наприклад, коміт з «feat» підвищує мінорну версію, а «fix» – патч-версію. Всі дії відбуваються у CI: після успішної збірки командою виводу npm publish відбувається безлюдний реліз. Завдяки цьому версії не

доводиться змінювати вручну – випуск проводиться автоматично при злитті в основну гілку.

3.3 Опис експериментального проєкту та рекомендації з використання систем контролю версій

Репозиторій Phantom0130/Trunk-based, доступний за посиланням [31], є прикладом реалізації стратегії магістральної розробки програмного забезпечення (TBD). Як уже було зазначено, цей підхід передбачає роботу з єдиною гілкою розробки (зазвичай main або trunk), до якої всі учасники проекту інтегрують свої зміни якнайчастіше, уникаючи довготривалих відгалужень. У репозиторії можна простежити ключові аспекти TBD, такі як короткоживучі feature-гілки, негайне вирішення конфліктів злиття та автоматизоване тестування для підтримки стабільності кодової бази.

У цьому випадку граф гілок (рис. 3.1) має вигляд лінійної історії з невеликими відгалуженнями, що швидко повертаються до основної лінії розробки, що є характерним для TBD. Також у репозиторії можна знайти Pull Requests, які використовуються для код-рев'ю перед злиттям змін до main, що підтверджує дотримання практик контролю якості.

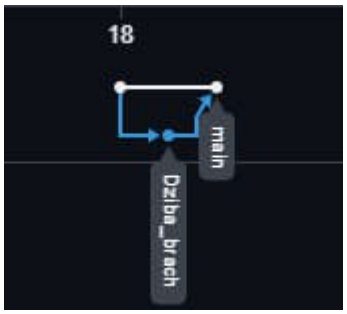


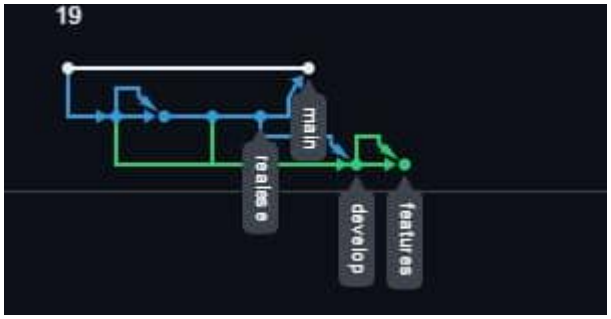
Рисунок 3.1 – Граф гілок для репозиторія з використанням магістральної розробки

Репозиторій Phantom0130/Git-Flow, доступний за посиланням [32], є прикладом використання методології управління гілками Git Flow. Ця модель

організації розробки програмного забезпечення передбачає чітке розділення гілок за функціональним призначенням, що сприяє ефективному контролю версій, особливо у великих проектах з планованими релізами.

Основним елементом структури репозиторію є дві стабільні гілки: main (або master) та develop (рис. 3.2а). Гілка main відображає стан коду, що відповідає останньому релізу, тоді як develop використовується для інтеграції нових функцій перед їх випуском. У репозиторії можна спостерігати, що всі зміни спочатку потрапляють у develop, а після завершення циклу розробки мердяться в main через механізм release-гілок. Візуалізацію впровадження стратегії галуження наведено на рис. 3.2б.

а)



б)

Рисунок 3.2 – Експериментальний репозиторій: а) базові гілки; б) додані гілки за стратегією GitFlow

Граф гілок репозиторія демонструє чітку ієрархію та контрольований процес злиття. Візуалізація показує, що feature-гілки відгалужуються від develop, а після завершення розробки інтегруються назад. Release-гілки утворюються з

develop і після тестування зливаються як у main, так і назад у develop для синхронізації змін. Гілки hotfix, у свою чергу, відходять від main, а після виправлення також повертаються до develop, запобігаючи розбіжностям у коді.

Фронтенд-розробка як складова сучасного веб-розвитку вимагає особливого підходу до організації процесів версіонування та автоматизації. Аналіз репозиторіїв Git-Flow та Trunk-Based дозволяє виявити ключові практики, специфічні для фронтенд-проєктів.

У репозиторії Git-Flow [32] реалізовано класичний підхід до управління фронтенд-розробкою. Особливу увагу приділено автоматизації процесів стилізації та валідації коду, що підтверджується наявністю конфігураційних файлів Prettier та ESLint. Історія комітів демонструє використання feature-гілок для розробки окремих UI-компонентів, де кожен елемент інтерфейсу проходить етап ізольованої розробки перед інтеграцією до гілки develop. Автоматизований пайплайн включає запуск юніт-тестів за допомогою Jest та візуальне тестування компонентів через Storybook, що особливо актуально для фронтенд-розробки. Trunk-Based репозиторій [31] пропонує альтернативний підхід до організації фронтенд-розробки. Основна гілка main містить завжди готовий до розгортання код, що досягається за рахунок суворого CI/CD пайплайну. Автоматизація включає інкрементальне тестування зміненого коду, що критично важливо для великих фронтенд-додатків. Практика короткоживучих гілок дозволяє швидко інтегрувати зміни в UI без ризику виникнення довгострокових конфліктів. Особливістю цього підходу є використання інструментів типу Chromatic для візуального регресійного тестування, що дозволяє виявляти проблеми зовнішнього вигляду компонентів на ранніх етапах.

Аналіз виявляє суттєві відмінності в підходах до управління статичними активами. У Git-Flow модель версіонування статичних ресурсів (CSS, зображень) прив'язана до релізного циклу, тоді як Trunk-Based підхід дозволяє оперативно оновлювати ці ресурси через безперервне розгортання. Обидва репозиторії демонструють інтеграцію з системами моніторингу якості коду (наприклад, SonarQube), що особливо актуально для підтримки великих фронтенд-проєктів.

Ключовим аспектом для фронтенд-розробки є організація процесу тестування. Git-Flow модель передбачає комплексне тестування на етапі release-гілки, тоді як Trunk-Based орієнтований на швидкі інкрементальні тести після кожного коміту. Обидва підходи мають свої переваги: перший забезпечує більш ретельну перевірку перед релізом, другий - оперативне виявлення помилок у UI-логіці.

На основі отриманого досвіду можна дати такі узагальнені поради. Слід використовувати Git (СКВ) у будь-якому фронтенд-проєкті. Розроблення нових функцій у окремих гілках значно спрощує організацію робочих процесів та ізолює нестабільний код від основної гілки. Навіть при магістральній розробці, де явна робота з гілками майже відсутня, «за кулісами» просто здійснюється інша форма галуження функціональних версій програмного забезпечення.

Автоматизовані збірки, тести і розгортання в GitHub Actions (або іншій CI-системі) мають бути стандартом для розроблення фронтенду. Налаштування конвеєра, який перевіряє кожен push/PR (установлює залежності, запускає тестування), гарантує, що на main потрапляє лише протестований код [29].

Рекомендованою практикою є впровадження конвенції оформлення комітів (наприклад, Conventional Commits) і використовувати лінтингові хуки. Автоматичні перевірки коду на етапі коміту (через Husky) допомагають підтримувати якість коду відпочатку. Регулярні code review та спільна робота передбачають визначення правил перевірки коду, обговорення змін у pull request-ах. Навчання команди основам Gitflow/GitHub Flow забезпечить консистентність і зрозумілість процесу.

Важливу роль відіграє також документування процесів: слід вести внутрішню документацію щодо стандартів комітів, галуження та безперервної інтеграції (наприклад, у README чи вікі проєкту). Це спрощує онбординг нових розробників та уніфікує практики команди.

Дотримання цих рекомендацій суттєво підвищить ефективність роботи та стійкість проєкту. Особливо важливо бачити Git не як «опцію», а як невід’ємну

частину розробки, оскільки відмова від системи контролю версій пов’язана з величезним ризиком, який жодна професійна команда не мала б приймати.

ВИСНОВКИ

У ході дослідження було проаналізовано особливості використання систем контролю версій у фронтенд розробці та розроблено методологію їх впровадження. Було виявлено, що системи контролю версій є невід'ємною частиною сучасної фронтенд розробки, забезпечуючи ефективну співпрацю розробників, відстеження змін, управління гілками розробки та автоматизацію процесів. Фронтенд-розробка має специфічні особливості, які впливають на використання систем контролю версій: різноманітність типів файлів, висока швидкість змін, компонентний підхід, активне використання зовнішніх бібліотек та необхідність синхронізації з бекенд-частиною.Git є найбільш оптимальною системою контролю версій для більшості фронтенд-проєктів завдяки своїй гнучкості, потужній системі гілок, широкій інтеграції з іншими інструментами та підтримці спільноти.

Вибір робочого процесу (workflow) має бути адаптованим до потреб конкретного проекту та команди. Для малих проектів підходить спрощена модель, для середніх та великих – Gitflow або магістральна розробка. Автоматизація є ключовим аспектом ефективного використання систем контролю версій. Використання хуків, CI/CD пайплайнів та інших інструментів автоматизації суттєво підвищує продуктивність розробки та якість коду. Правильно налаштована інтеграція з іншими інструментами фронтенд розробки (менеджерами пакетів, системами CI/CD, системами управління проектами) забезпечує ефективну екосистему розробки. Належне навчання команди та документування процесів є важливими факторами успішного впровадження системи контролю версій.

Впровадження системи контролю версій має позитивний вплив на якість коду, продуктивність розробки та командну роботу, що підтверджується результатами практичного впровадження. Результати дослідження можуть бути використані при впровадженні систем контролю версій у фронтенд-проєктах різного масштабу, від невеликих стартапів до великих корпоративних додатків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 2024 Stack Overflow Developer Survey. Stack Overflow Insights - Developer Hiring, Marketing, and User Research. URL: https://survey.stackoverflow.co/2024/ (дата звернення: 20.02.2025).

2. State of Frontend 2024. *TSH*. URL: https://tsh.io/state-of-frontend (дата звернення: 20.02.2025).

3. Securities.io — Технології та фінанси. URL: https://securities.io/ru/?offset=792&exchange=nasdaq (дата звернення: 20.02.2025)

4. Git – система контролю версій. Git. URL: https://git-scm.com (дата звернення: 20.02.2025).

5. Collaboration software for software, IT and business teams. Collaboration software for software, IT and business teams. URL: https://atlassian.com (дата звернення: 22.02.2025).

6. dsp. *dsp*. URL: https://experimentalworks.net/posts/ (дата звернення: 27.03.2025).

7. Initial Commit - Quality resources and tools for developers. *Initial Commit*. URL: https://initialcommit.com (дата звернення: 27.03.2025).

8. Korvage Information Technology. Differentiate Between Centralized & Distributed Version Control System. LinkedIn: Log In or Sign Up. URL: https://www.linkedin.com/pulse/differentiate-between-centralized-distributed-version-control-ejv2c (дата звернення: 01.04.2025).

9. DevOps Solutions for Innovation at Scale | Enterprise DevOps Tools. *Perforce*. URL: https://perforce.com (дата звернення: 03.04.2025).

10. Collaboration Tools for Team & Project Management | Nulab. *Nulab*. URL: https://nulab.com (дата звернення: 05.04.2025).

11. Package.json vs Package-lock.json. *Atatus Blog - For DevOps Engineers, Web App Developers and Server Admins.* URL: https://www.atatus.com/blog/package-json-vs-package-lock-json/ (дата звернення: 05.04.2025).

12. Building Micro-Frontends with React, Vue, and Webpack 5. URL: https://techwards.co/react-vue-micro-frontend-application-using-webpack-5-module-federation/ (дата звернення: 20.04.2025).

13. Demkovych I. Everything You Need To Know About Git Flow Model. Geniusee. URL: https://geniusee.com/single-blog/everything-you-need-to-know-about-git-flow-branch-model (дата звернення: 07.05.2025).

14. Mallikarjunaiah M., Mendy M. Explaining Trunk Based Development. URL: https://www.travis-ci.com/blog/explaining-trunk-based-development/ (дата звернення: 07.04.2025).

15. DEV Community. DEV Community. URL: https://dev.to (дата звернення: 07.04.2025).

16. Toptal. Toptal. URL: https://www.toptal.com/ (дата звернення: 16.04.2025).

17. The forefront of innovation | Flatirons. The forefront of innovation | Flatirons. URL: https://flatirons.com (дата звернення: 07.05.2025).

18. DevOps - The Web's Largest Collection of DevOps Content. DevOps.com. URL: https://devops.com (дата звернення: 14.04.2025).

19. PixelFreeStudio Blog. PixelFreeStudio Blog -. URL: https://blog.pixelfreestudio.com (дата звернення: 15.04.2025).

20. Web Hosting for Everyone - DreamHost. DreamHost. URL: https://dreamhost.com (дата звернення: 14.04.2025).

21. Conventional Commits. *Conventional Commits*. URL: https://conventionalcommits.org (дата звернення: 15.04.2025).

22. GitLab Docs. *GitLab Docs*. URL: https://docs.gitlab.com (дата звернення: 15.04.2025).

23. GitHub — репозиторії та CI/CD. URL: https://github.com (дата звернення: 13.05.2025)

24. React. React. URL: https://react.dev/ (дата звернення: 13.05.2025).

25. Vite | Next Generation Frontend Tooling. Vite. URL: https://vite.dev (дата звернення: 15.05.2025).

26. Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. *Tailwind CSS*. URL: https://tailwindcss.com (дата звернення: 20.05.2025).

27. Find and fix problems in your JavaScript code - ESLint - Pluggable JavaScript Linter. Eslint. URL: https://eslint.org (дата звернення: 10.05.2025).

28. typicode. URL: https://typicode.github.io (дата звернення: 20.05.2025).

29. GitHub.com Help Documentation. GitHub Docs. URL: https://docs.github.com (дата звернення: 20.05.2025).

30. Oyebisi J. Feature Flags: A Technique for Modifying Application Behavior without Altering Code. URL: https://medium.com/@oyebisijemil_41110/feature-flags-a-technique-for-modifying-application-behavior-without-altering-code-309801026a2f (дата звернення: 01.05.2025).

31. GitHub - Phantom0130/Trunk-based. GitHub. URL: https://github.com/Phantom0130/Trunk-based (дата звернення: 30.05.2025).

32. GitHub - Phantom0130/Git-Flow. GitHub. URL: https://github.com/Phantom0130/Git-Flow (дата звернення: 30.05.2025).

ДОДАТКИ

Додаток А – Шаблони та правила комітів

Для поліпшення читабельності історії комітів вводяться правила оформлення повідомлень. Зокрема, може застосовуватися конвенція *Conventional Commits*, де заголовок коміту починається з типу зміни (feat:, fix:, chore: тощо). Наприклад:

makefile

КопіюватиРедагувати

feat: додано модуль керування товарами

fix: виправлено відображення ціни при оновленні

chore: оновлено залежності до останніх версій

Такий формат дозволяє швидко зрозуміти характер змін і при потребі автоматизовано формувати список змін. Для контролю форматування повідомлень можна використати Husky разом із лінтером коміт-повідомлень (наприклад, commitlint). Офіційний сайт Husky рекомендує ініціалізувати хук одним рядком:

bash

КопіюватиРедагувати

npm install --save-dev husky

npx husky install

npm pkg set scripts.prepare "husky install"

npx husky add .husky/commit-msg "npx commitlint --edit $1"

Це створює скрипт .husky/commit-msg, який перевірятиме повідомлення перед кожним комітом. Зазвичай у хуку commit-msg додають виклик commitlint або власний скрипт для валідації формату. Таким чином, команда забезпечує автоматичну перевірку стилю комітів.

Додаток Б – Налаштування Git Hooks (Husky)

За допомогою Husky додаються Git-hook-и, які виконуються на різних етапах. Команда npx husky init автоматично створює каталог .husky і шаблон хука pre-commit[28]. Наприклад:

bash

КопіюватиРедагувати

```
npm install --save-dev husky
npx husky install
npx husky add .husky/pre-commit "npm run lint"
```

Це створить файл .husky/pre-commit із командою npm run lint. У даному випадку перед кожним комітом буде запускатися ESLint (або інший скрипт). Як показано в документації, достатньо двох рядків у шелл-скрипті для перевірки змінених файлів з Prettier або ESLint. Таким чином, Husky гарантує, що код у репозиторії завжди відповідає стандартам стилю та проходить всі тести перед фіксацією.

Додаток В – Налаштування Git Hooks (Husky)

На рівні репозиторію створюється конфігурація CI/CD за допомогою GitHub Actions. Наприклад, файл .github/workflows/ci.yml може виглядати так:

yaml

КопіюватиРедагувати

```
name: CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Use Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
      - run: npm ci
      - run: npm run build
      - run: npm test
```

Така конфігурація запускає збірку і тести при кожному пуші чи pull request у гілку main. Крім того, GitHub аналізує проєкт і пропонує шаблони робочих процесів для конкретних технологій. Наприклад, для Node.js він автоматично радить шаблон workflow, який встановлює пакети та запускає тести. Такий підхід

спрощує налаштування інтеграції, а результати тестів відображаються прямо у pull request, що полегшує виявлення помилок до злиття коду.

Загалом, етап впровадження СКВ включає послідовні кроки: створення репозиторію (git init), налаштування .gitignore, реалізацію гілкового робочого процесу (feature-бренчинг), введення стандарту комітів, додавання Git-хуків через Husky, та налаштування автоматизації через CI/CD (GitHub Actions). Кожен із цих кроків підвищує організованість розробки та якість коду.