

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«МИРЭА – Российский технологический университет»**

**РТУ МИРЭА**

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных технологий

# Отчет по практической работе №6

по дисциплине «Сопровождение программных систем»

Тема: «Система контроля версий Git»

|  |  |
| --- | --- |
| **Выполнил:**  Студент группы ИКБО-11-22 | Гришин А. В. |
| **Проверил:** | Рысин М. Л. |

Москва, 2025 г.

**Цель работы**

Познакомиться с возможностями системы контроля версий Git.

# Ход выполнения практической работы

Перед началом основных этапов была выполнена проверка предварительных условий. В частности, включено отображение скрытых файлов и папок (рисунок 1).

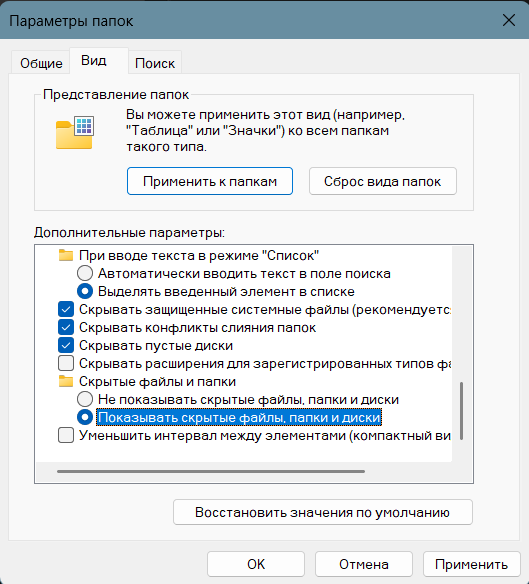


Рисунок 1 – Отображение скрытых папок

Также подтверждено наличие установленного интерпретатора Python (рисунок 2) и редактора Visual Studio Code (рисунок 3).

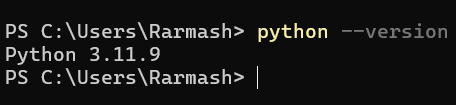


Рисунок 2 – Проверка установленной версии Python



Рисунок 3 – Запуск Visual Studio Code

Проведена проверка установки расширения Pylance для поддержки работы с Python-скриптами (рисунок 4), а также наличие установленной версии Git (рисунок 5).

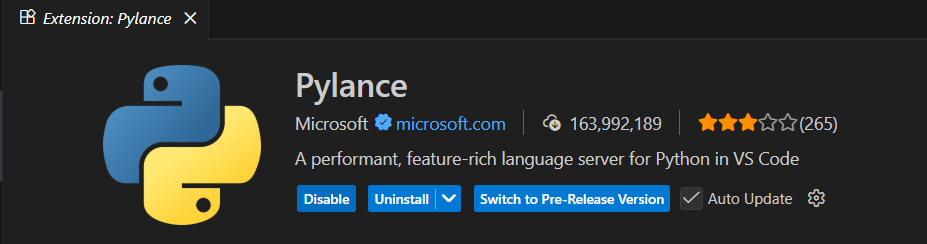


Рисунок 4 – Установленное расширение Pylance

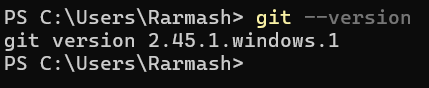


Рисунок 5 – Проверка установленной версии Git

Поменяем предустановленные параметры Git. Результат смены необходимых настроек изображен на рисунке 6.

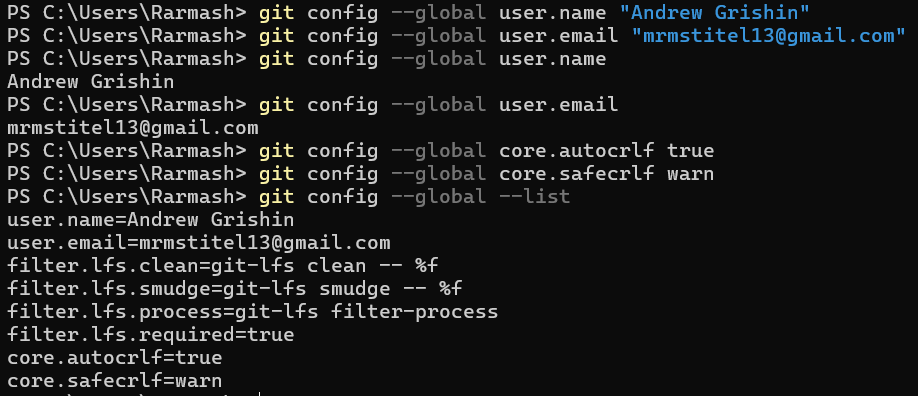


Рисунок 6 – Проверка и изменение настроек Git

Установлены глобальные настройки, обеспечивающие корректное отображение символов в любой кодировке и использование ветки «main» по умолчанию (рисунок 7).

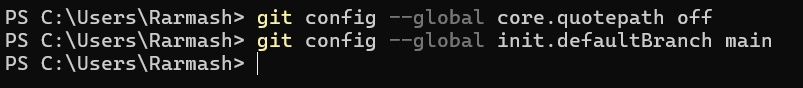


Рисунок 7 – Команды для избежания нечитаемых строк и названия основной ветки по умолчании

Найдём файл .gitconfig, в котором хранятся глобальные настройки Git. Место расположения этого файла показано на рисунке 8. Содержимое файла показано на рисунке 9.



Рисунок 8 – Файл с глобальными настройками Git

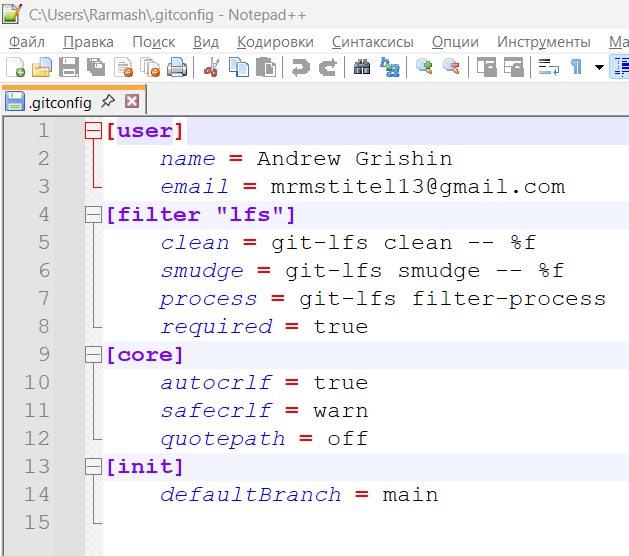


Рисунок 9 – Содержимое файла .gitconfig

С помощью команд git help -g (рисунок 10) и git help commits (рисунок 11) получена справочная информация по доступным командам Git, включая переход в браузер для просмотра подробного описания конкретной команды commits.

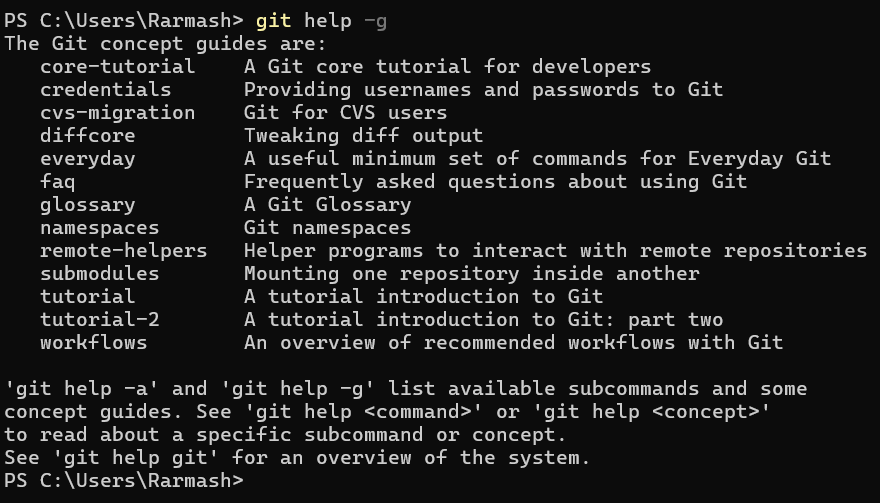


Рисунок 10 – Подсказка по всем доступным командам

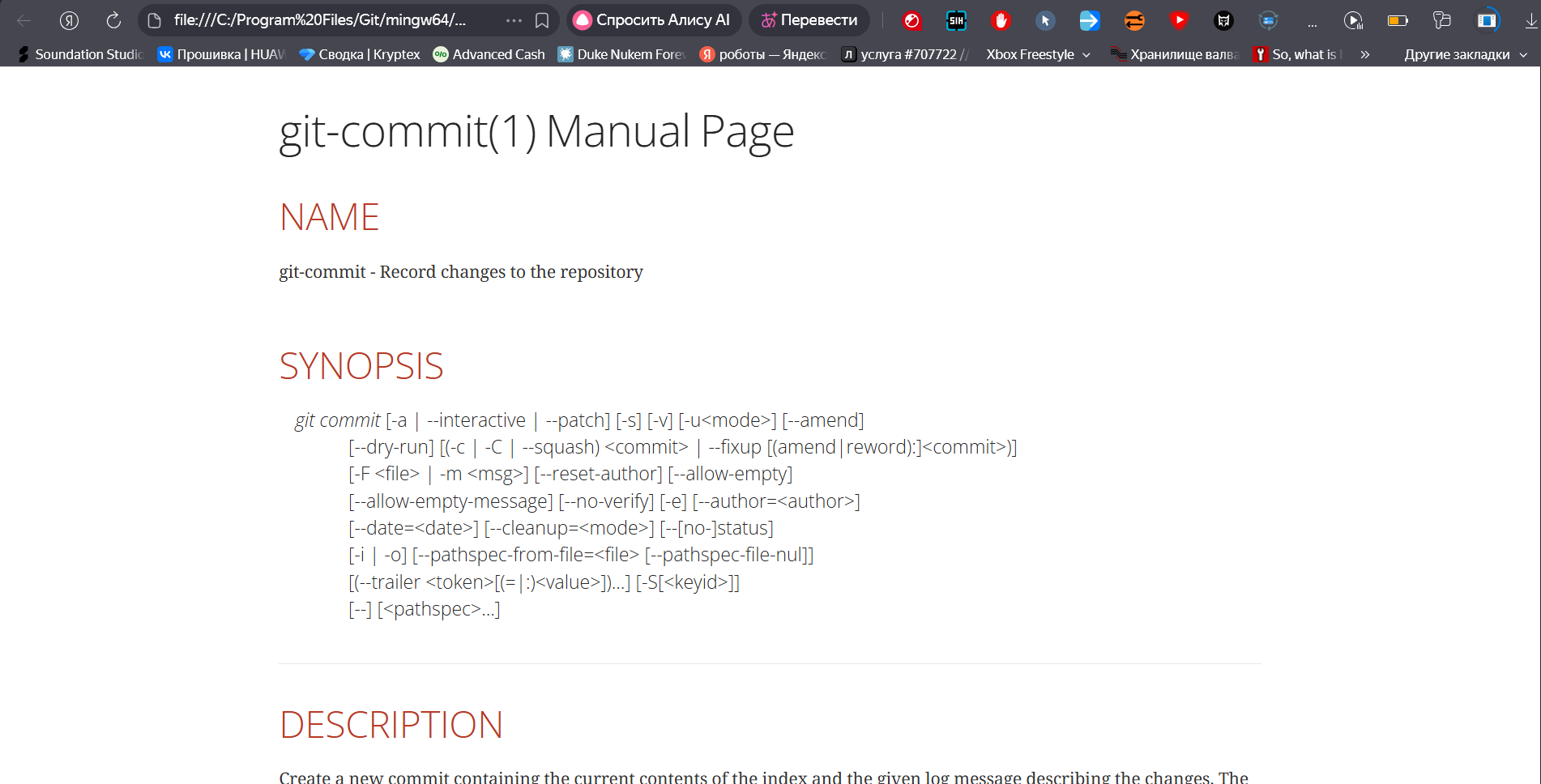


Рисунок 11 – Вывод второй команды

Была создана папка с инициалами исполнителя, в ней инициализирован новый репозиторий командой git init (рисунок 12). В Visual Studio Code откроем созданный каталог. Добавим файл README.md (рисунок 13).

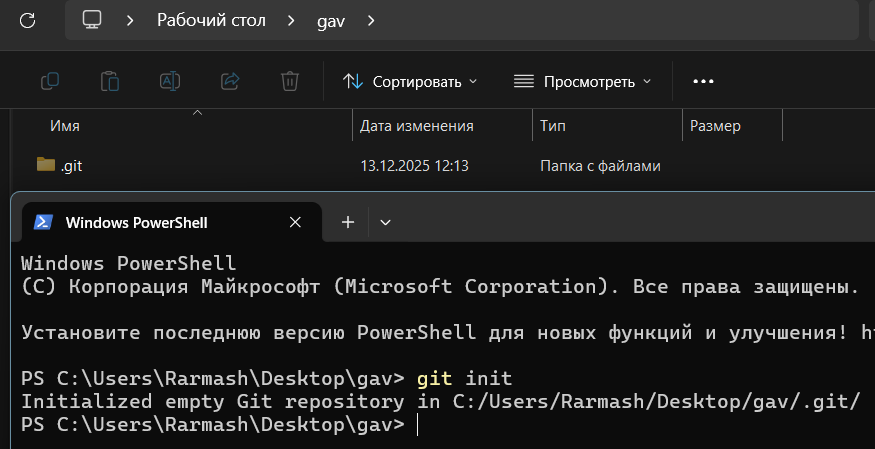


Рисунок 12 – Создание папки с инициалами и инициализация репозитория

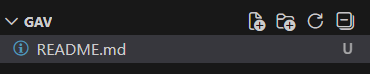


Рисунок 13 – Создание README файла в папке с инициалами

Выполним git status, чтобы увидеть, что новый файл ещё не отслеживается. Добавим файл в индекс командой git add README.md и снова проверим статус — теперь файл готов к коммиту (рисунок 14-15).

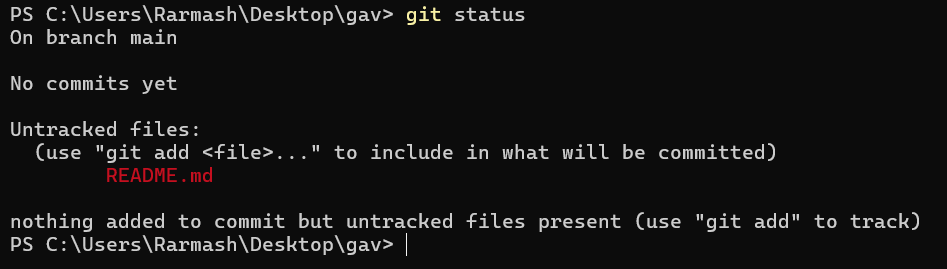


Рисунок 14 – Проверка результата git status

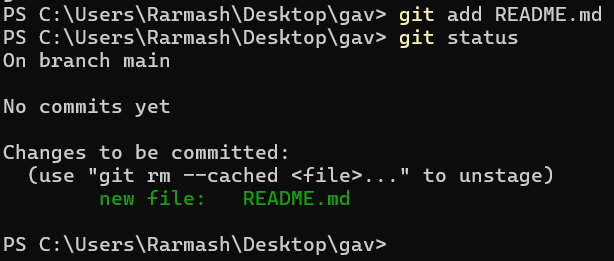


Рисунок 15 – Состояние репозитория после добавления в индекс файла README.md

Внесём первые изменения в README.md, сохраним файл и снова выполним git status: Git отметит модификацию (рисунок 16).

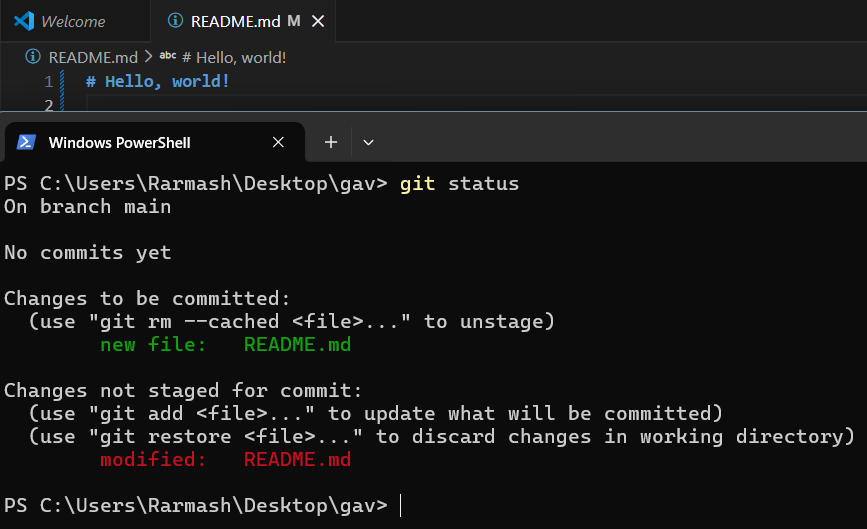


Рисунок 16 – Изменения в файле README.md и результат выполнения команды git status

Добавим изменённый файл в индекс и сделаем первый коммит с осмысленным сообщением (рисунок 17-18).

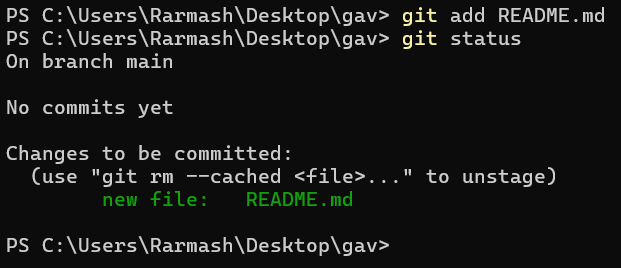


Рисунок 17 – Состояние репозитория после добавления в индекс файла README.md

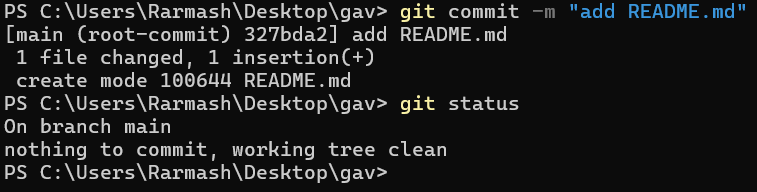


Рисунок 18 – Состояние репозитория после выполнения коммита

Затем добавим в файл строку «I study at MIREA», сохраним, добавим в индекс и выполним второй коммит (рисунок 19).

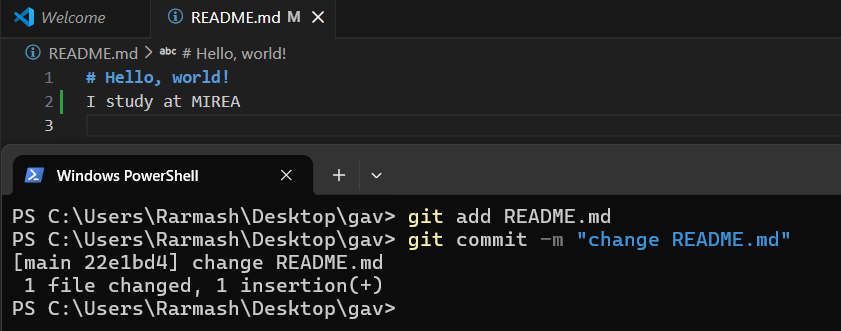


Рисунок 19 – Изменение файла README.md и коммит изменений

Команда git log выводит историю коммитов в обратном хронологическом порядке, предоставляя информацию о каждом коммите, включая хэш, автора, дату и сообщение. Эта команда является основным инструментом для просмотра истории изменений в репозитории и может быть использована с различными флагами для фильтрации, форматирования и ограничения вывода. Выполнение команды показано на рисунке 21.



Рисунок 21 – История коммитов проекта

Чтобы посмотреть всю информацию о коммите, выполним команду git log без флага. Результат выполнения команды показан на рисунке 22.

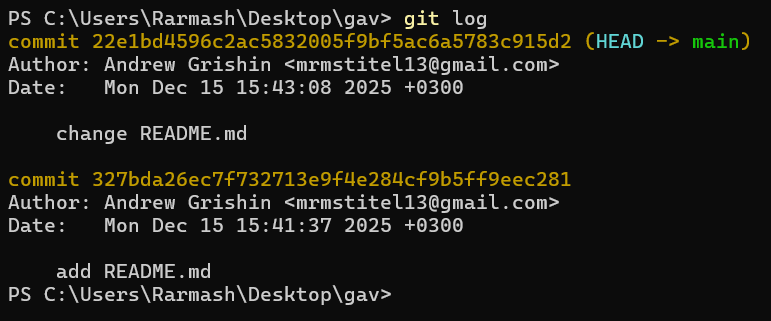


Рисунок 22 – Полная история коммитов проекта

Сделаем коммит с заведомо неверным сообщением, проверим его в истории, а затем исправим сообщение последнего коммита с помощью git commit --amend -m "новое правильное сообщение". Убедимся, что история обновилась корректно (рисунок 23).

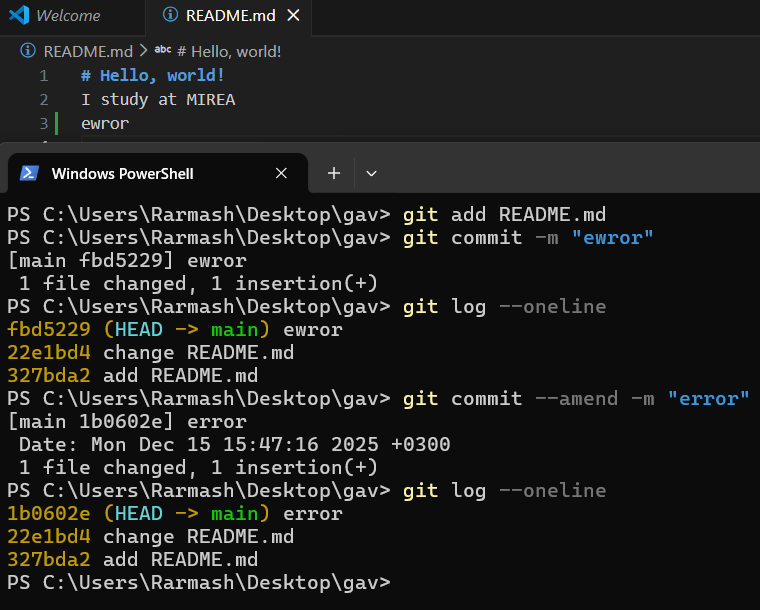


Рисунок 23 – Создание коммита с ошибкой в сообщение и ее последующее исправление

Создадим новый файл errors.md, заполним его произвольным содержимым, добавим в индекс и закоммитим. С помощью git log найдём хэш коммита, сделанного до добавления этого файла (рисунок 24).

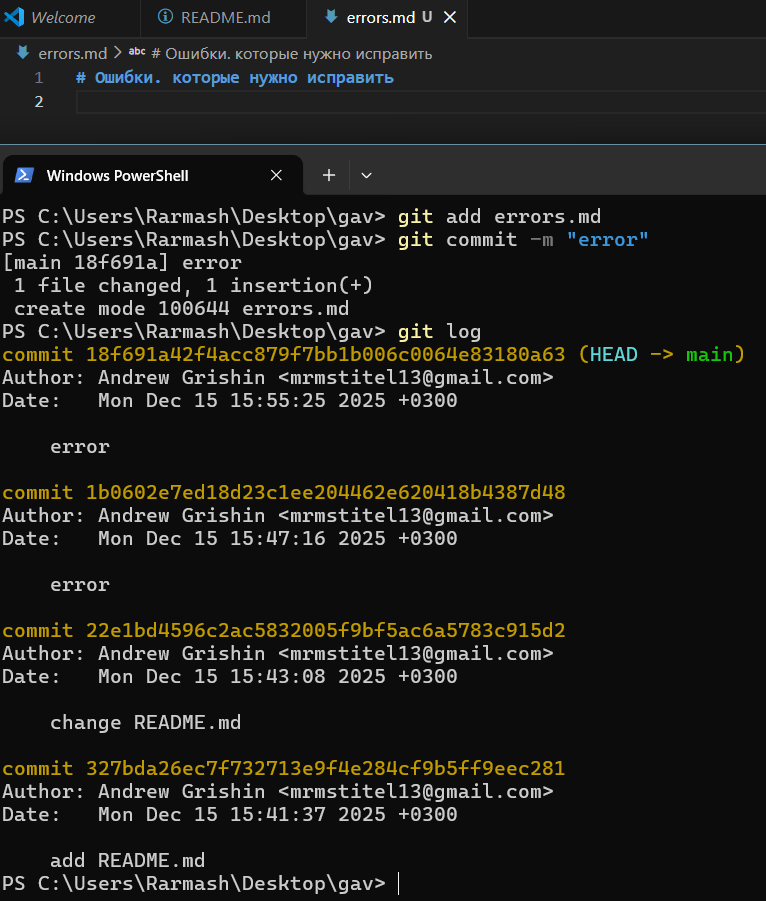


Рисунок 24 – Создание коммита с новым файлом нахождение хеша этого коммита

Переключимся на этот коммит командой git checkout <хэш> — файл errors.md исчезнет из рабочей директории (рисунок 25).

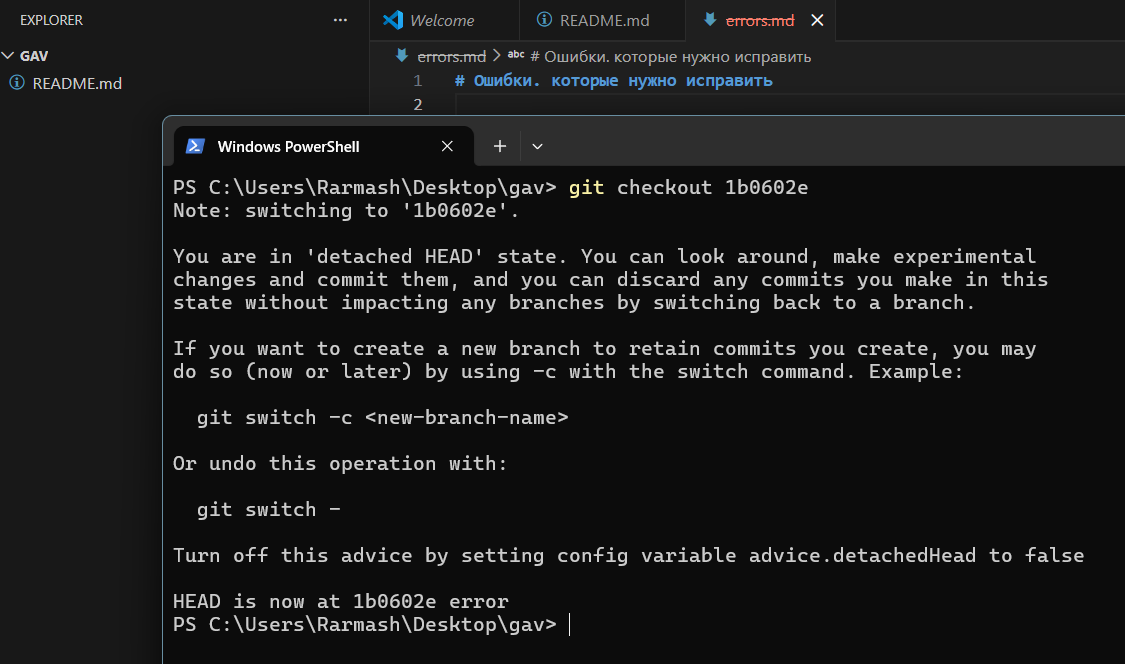


Рисунок 25 – Откат к предпоследнему коммиту

Выполним команду git log для проверки истории коммитов и увидим, что мы действительно откатились на один коммит назад. С помощью добавления флага --all мы можем увидеть всю историю проекта вне зависимости от того, на каком коммите мы сейчас находимся (рисунок 26).

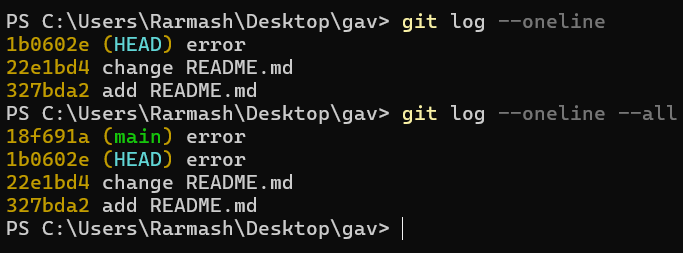


Рисунок 26 – История проекта до текущего коммита и вся история целиком

Вернёмся на последнюю версию командой git checkout main (рисунок 27).

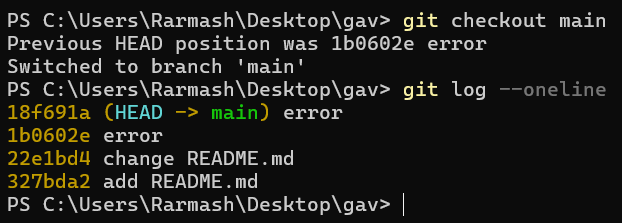


Рисунок 27 – Возвращение к верхнему снимку проекта

GitHub – самая популярная в мире площадка для облачного хранения кода и совместной командной разработки. Но у неё, конечно, есть множество альтернатив, например платформы GitLab, Bitbucket, а также собственных серверных решений. Войдем в свой аккаунт GitHub. Попадём в личный профиль (рисунок 28).

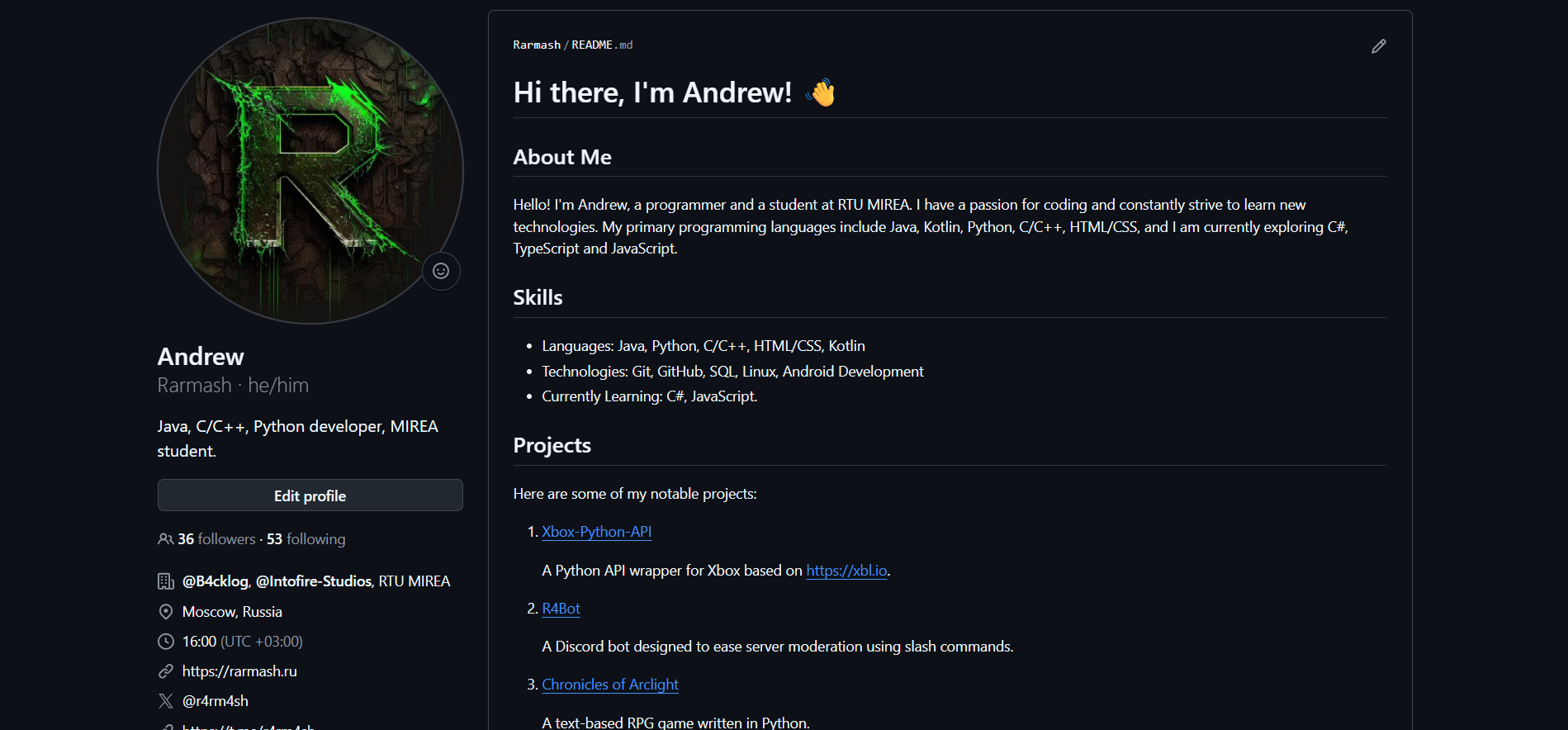


Рисунок 28 – Главная страница профиля GitHub после входа

Откроем терминал и выполним команду ssh-keygen. Программа спросит, куда сохранить файл с ключом и предложит дважды ввести пароль для этого ключа. На этом этапе оставим все настройки по умолчанию (просто нажмем несколько раз Enter). На рисунке 29 показан результат генерации SSH ключа.

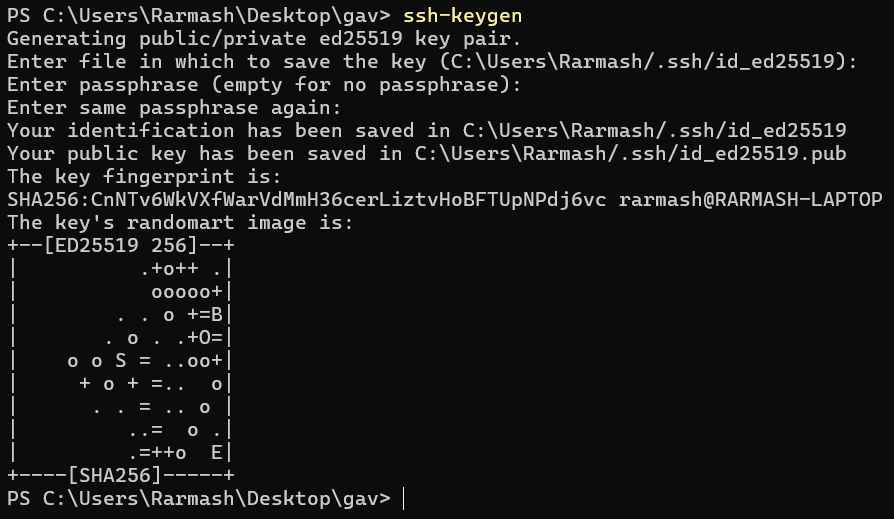


Рисунок 29 – Результат генерации SSH ключа

С помощью команды cat ~/.ssh/id\_ed25519.pub выведем сгенерированный ключ. Добавим выданный ключ в настройках нашего GitHub. Результат показан на рисунке 30.

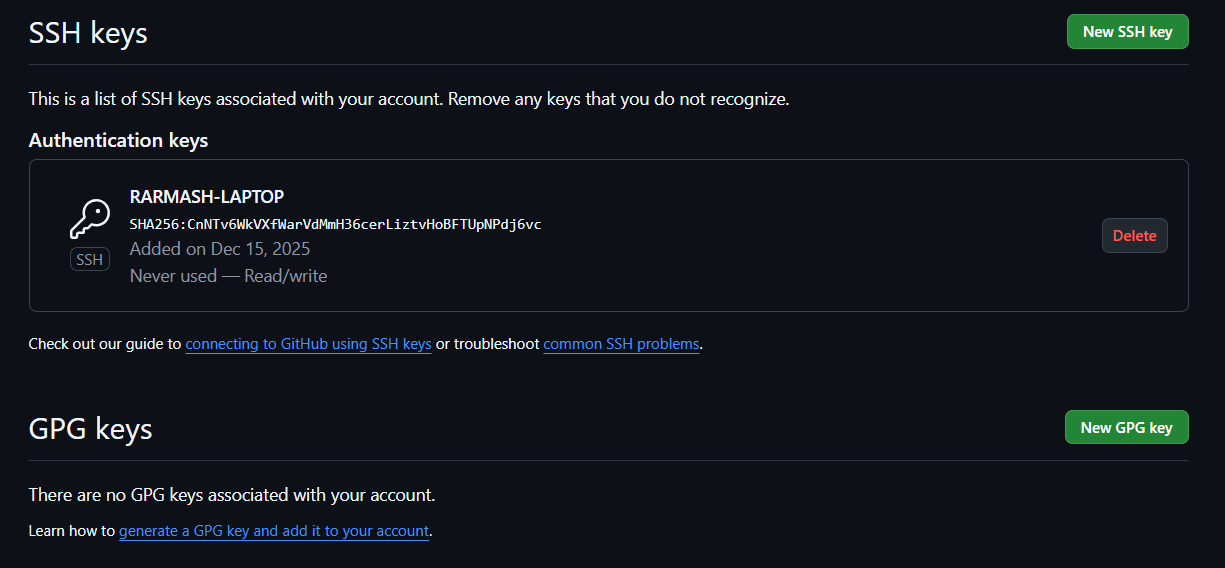


Рисунок 30 – Результат добавления SSH ключа в Github

Создадим новый репозиторий с названием gav, т.е. с именем папки, созданной ранее в рамках практической работы. Процесс создания показан на рисунке 31.

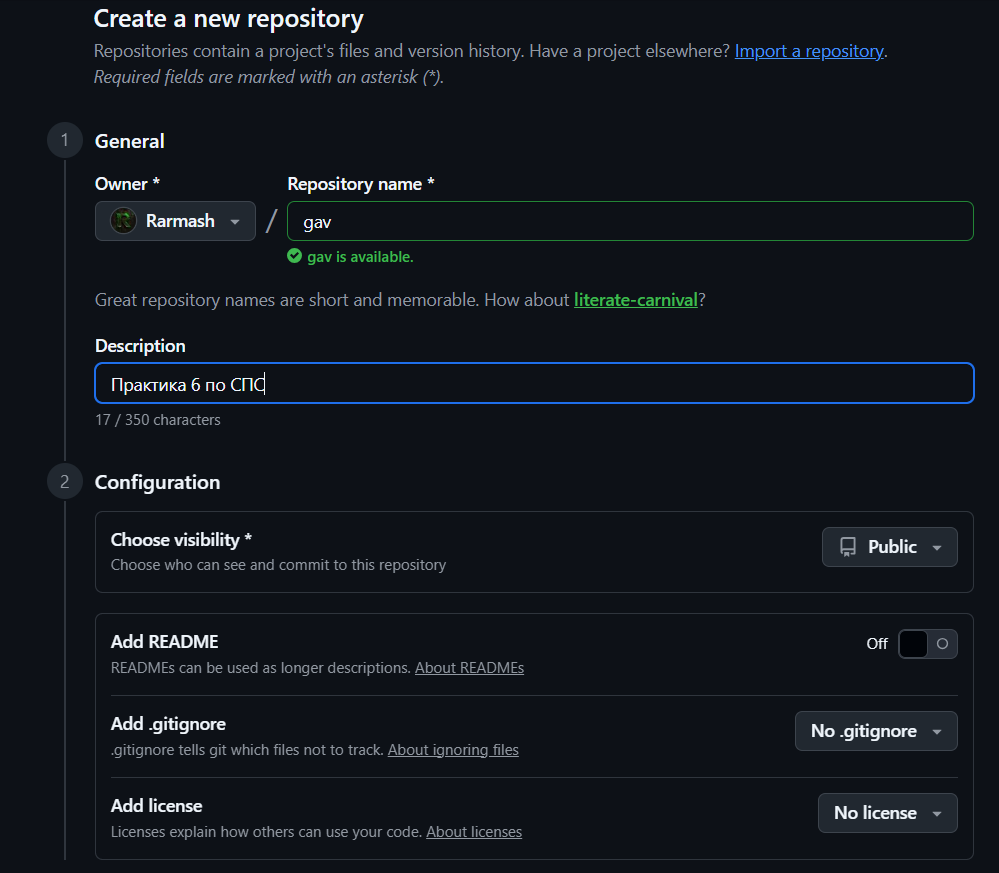


Рисунок 31 – Создание нового репозитория

Проверим, что локальный репозиторий все еще не привязан ни к какому удаленному с помощью команду git remote -V. Так как Git ничего не ответил, можем сделать вывод, что репозиторий действительно не привязан. Привяжем его к нашему только что созданному репозиторию с помощью команды git remote add origin git@github.com:Rarmash/gav.git (рисунок 32).

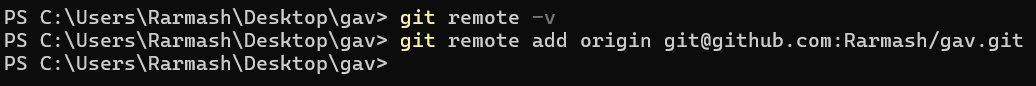


Рисунок 32 – Привязка локального репозитория к удаленному

Отправим наш проект в удаленный репозиторий с помощью команды “git push -u origin main”. Результат выполнения команды показан на рисунке 33.

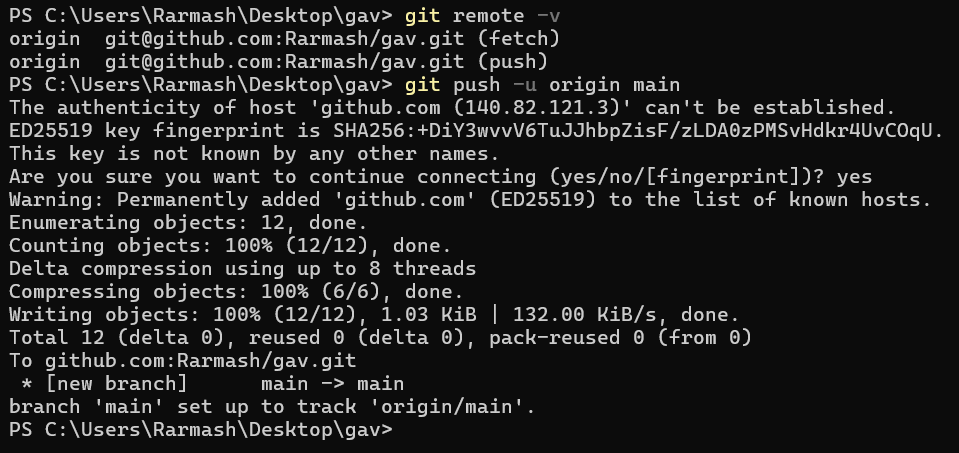


Рисунок 33 – Отправка проекта в удаленный репозиторий

Вернемся в браузер и убедимся, что проект действительно отправился на удаленный репозиторий (рисунок 34).

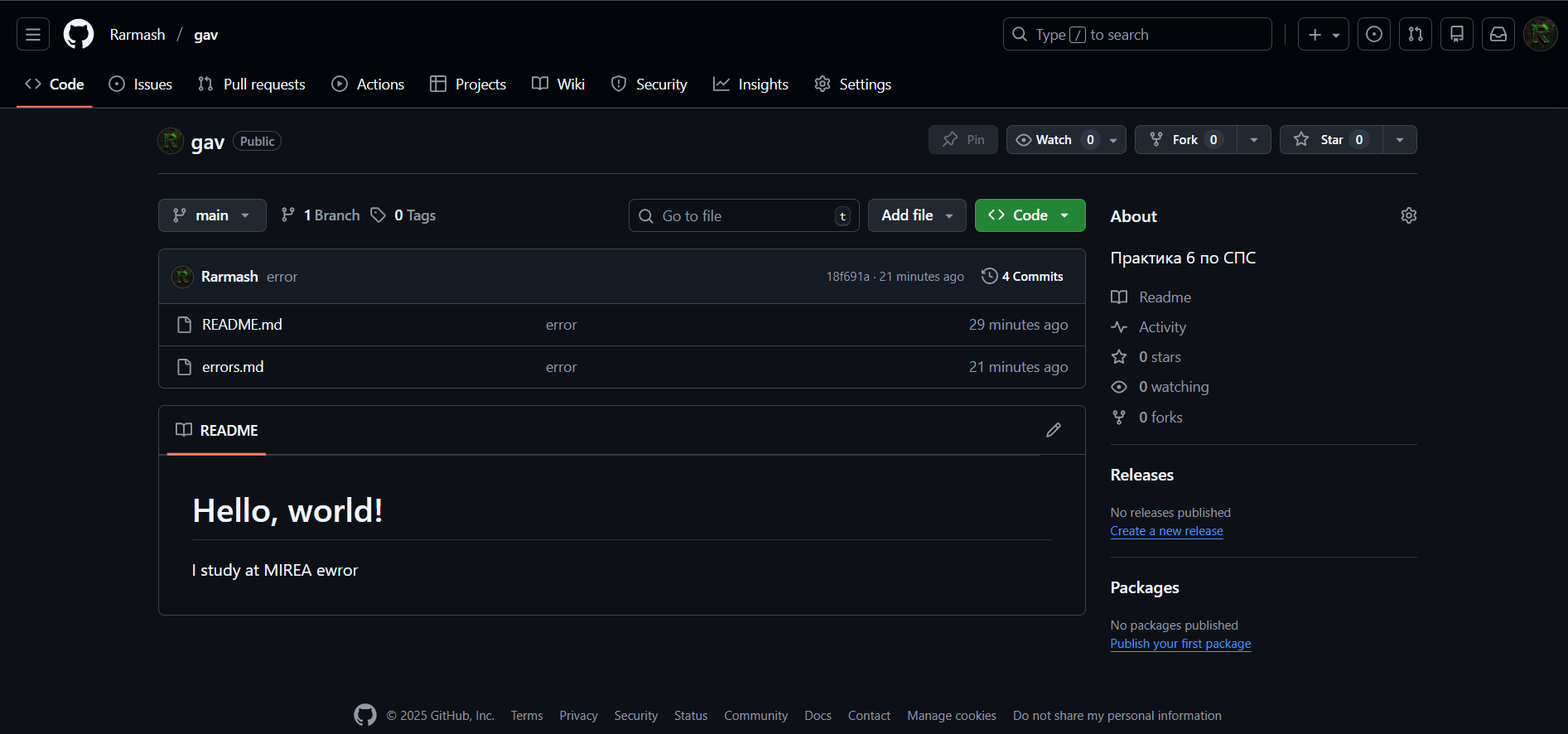


Рисунок 34 – Главная страница репозитория

По нажатию на историю коммитов можно увидеть всю историю проекта ровно так же, как и в терминале (рисунок 35).

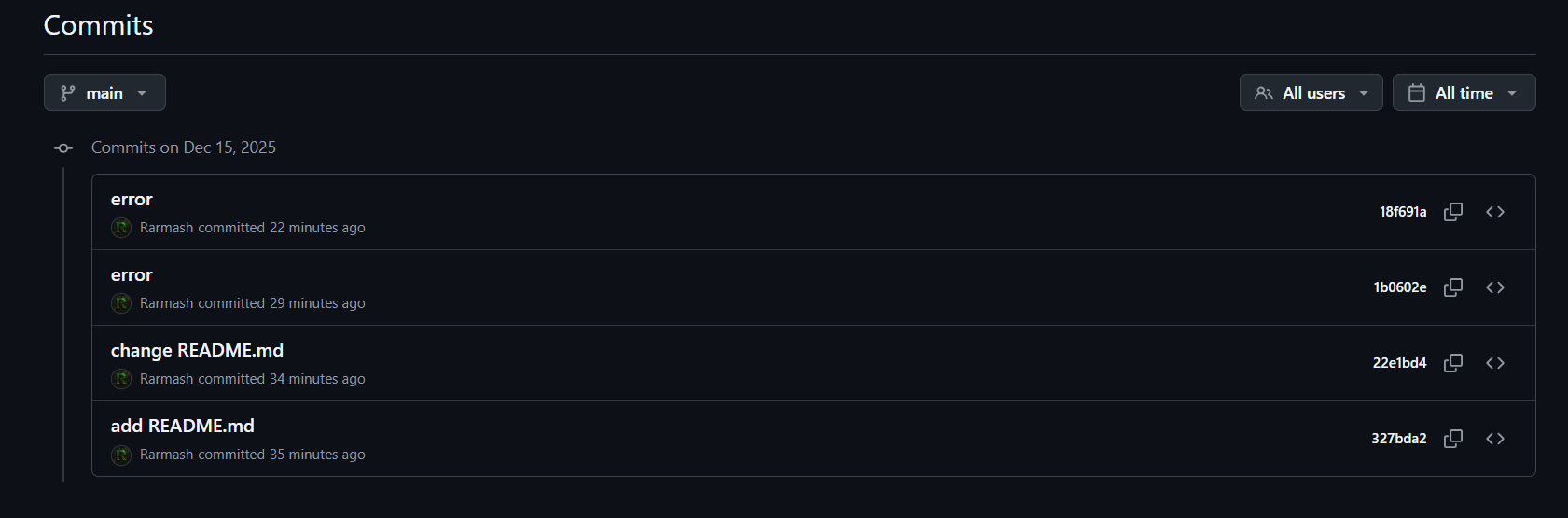


Рисунок 35 – История коммитов репозитория

Fork (форк) – механизм, который позволяет создать копию опубликованного на GitHub репозитория и работать с ним самостоятельно. При создании форка репозиторий копируется в вашу учётную запись на GitHub. Это позволяет вносить изменения в свою копию проекта без ущерба для основного репозитория.

Откроем в браузере ссылку https://github.com/netology[-](https://github.com/netology-code/git-2-fork)code/git[-2-fork.](https://github.com/netology-code/git-2-fork) В верхнем правом углу найдем кнопку с надписью Fork и нажмем на неё. GitHub уточнит у нас, куда сохранить копию (рисунок 36). Нажмем на кнопку Create fork. На короткое время появится окно с надписью «Forking netology-code/git-2-fork». Потом страница обновится, и мы увидим уже свою копию проекта (рисунок 37).

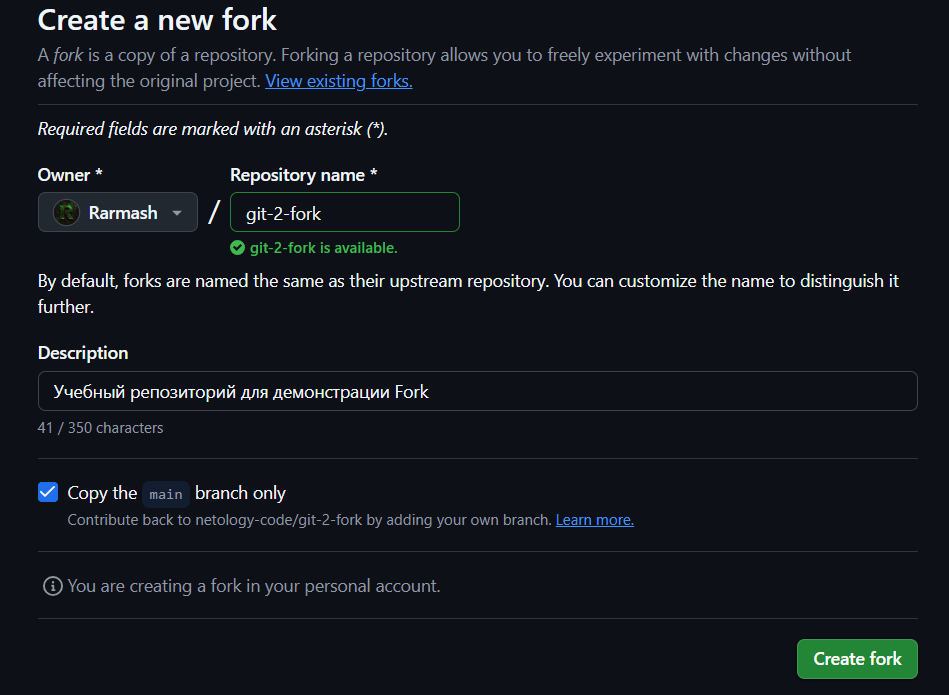


Рисунок 36 – Создание форка репозитория

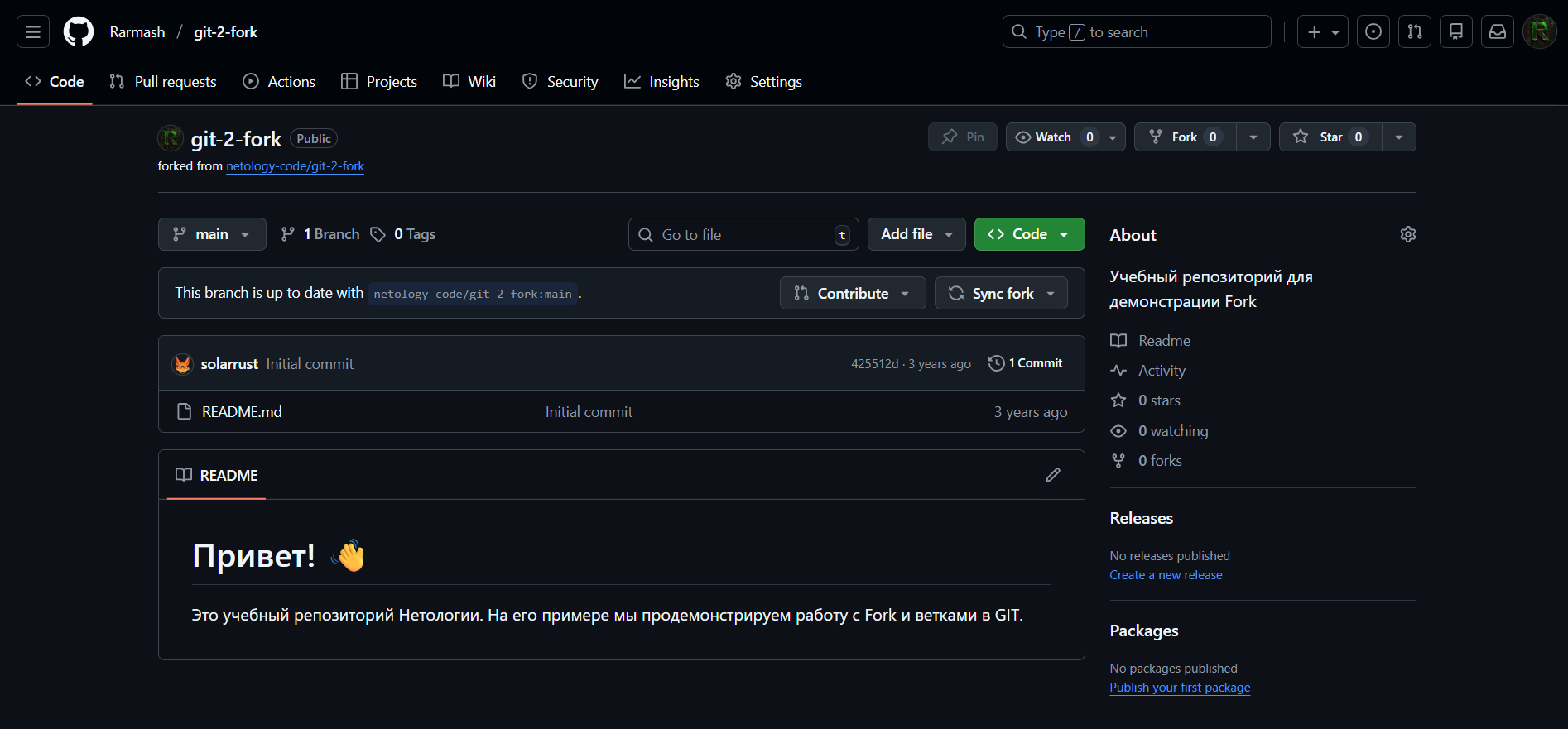


Рисунок 37 – Созданный форк репозитория

На странице форк-репозитория нажмем кнопку Code. В появившемся окне нажмём на вкладку SSH и скопируем адрес из поля ниже. На ПК перейдем на рабочий стол и откроем для него терминал. В терминале напишем команду git clone и вставим только что скопированную ссылку. Нажмем Enter и дождемся окончания операции, после чего перейдем в склонированный репозиторий (рис. 38).

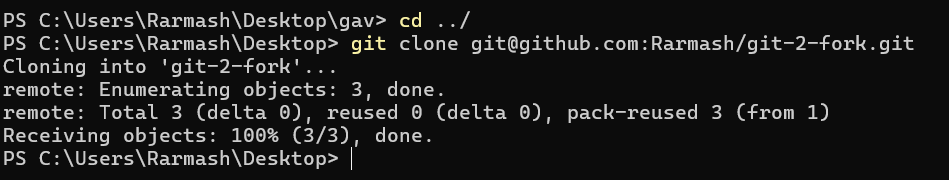


Рисунок 38 – Копирование удаленного репозитория и переход в него

Ветка – опция в Git, которая позволяет создавать параллельно реализуемые клоны проекта, когда:

− над проектом работает несколько человек;

− каждый делает свою задачу;

− чтобы не мешать друг другу, каждый работает в своей ветке; − когда задача готова, то всё сливается в основную ветку.

Использование веток:

а) помогает избежать поломок в основном проекте, пока функция не

готова;

б) избавляет от конфликтов, когда несколько человек исправляют одну

и ту же часть по-разному.

Для работы с ветками существует команда git branch. Если набрать её без дополнительных параметров, то Git покажет список всех локальных веток проекта. Наберем команду и проверим вывод (рисунок 39).

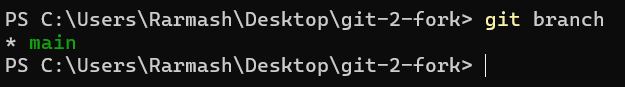


Рисунок 39 – Выполнение команды git branch

Создадим новую ветку с помощью команды git branch <название ветки> и проверим, что ветка создалась с помощью команды git branch. Перейдем на новую ветку с помощью команды git checkout new-branch и проверим, что мы действительно переключились на новую ветку. Результат выполнения описанных команд показан на рисунке 40.

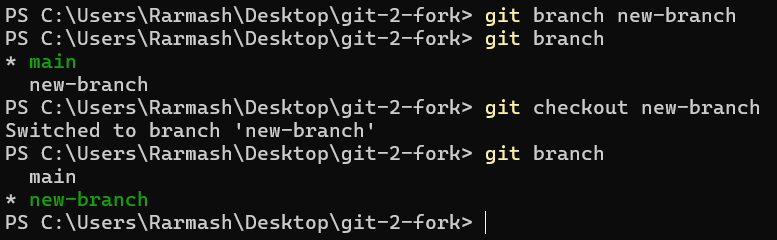


Рисунок 40 – Создание новой ветки и переключение не нее

Откроем в Visual Studio Code папку git-2-fork. В файле README.md ниже допишем что-нибудь. Теперь файл README.md в ветке new-branch отличается от файла README.md в основной ветке main (рисунок 41).

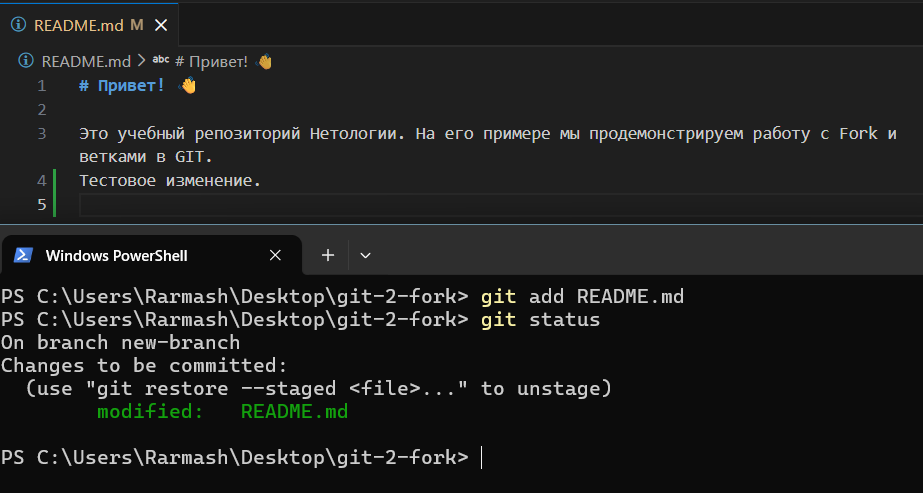


Рисунок 41 – Изменение файла README.md в ветке new-branch

Сделаем коммит с сообщением, соответствующем изменениям и отправим его в удаленный репозиторий (рисунок 42).

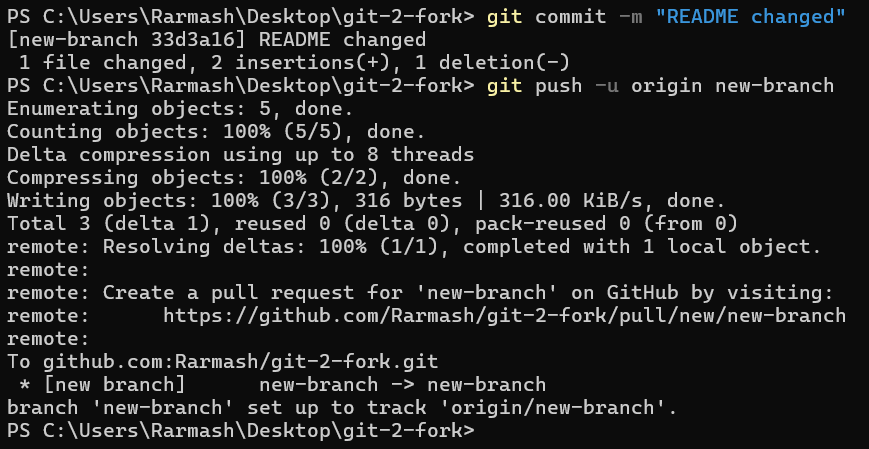


Рисунок 42 – Выполнение и отправка коммита на удаленный сервер

На сайте GitHub на странице проекта переключимся на ветку new-branch и убедимся в корректном выполнении коммита. Обновленное содержание README.md показано на рисунке 43.

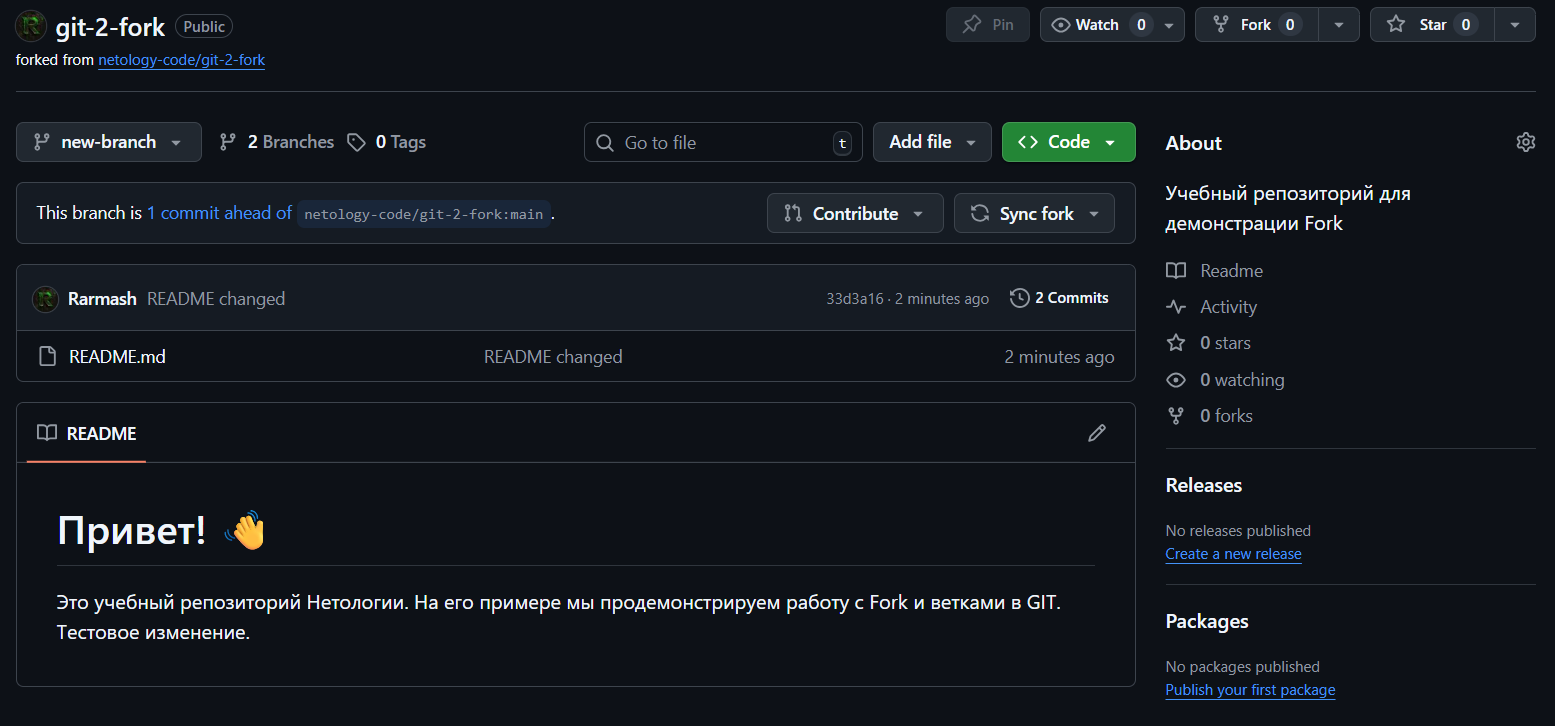


Рисунок 43 – Обновленное содержание README.md на ветке new-branch

Просмотрите историю новой ветки в терминале. Результат выполнения команды “git log new-branch --graph –oneline” показано на рисунке 44.

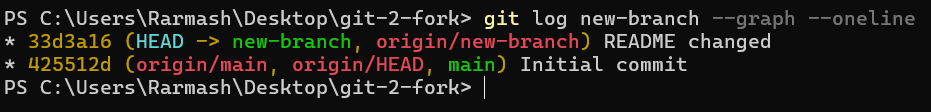


Рисунок 44 – Результат выполнения команды git log new-branch --graph –oneline

Переключимся на основную ветку и сольем изменения с помощью команды merge. Выгрузим изменения на удаленный репозиторий с помощью команды push (без -u origin main). Это дополнение нужно только первый раз, когда отправляете новую ветку в GitHub. Git уже знает, что и куда отправлять. Результат выполнения слияния и пуша показан на рисунке 45.

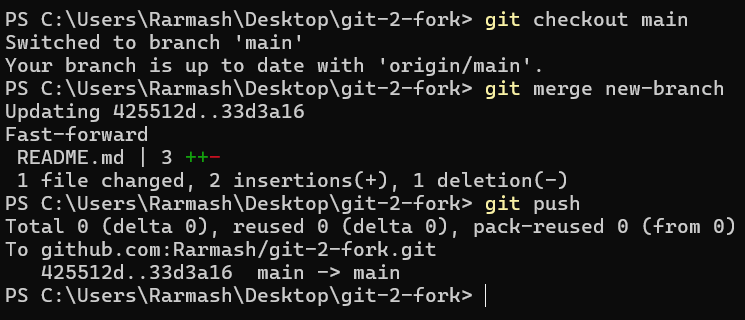


Рисунок 45 – Результат слияния веток

Проверим результат слияния, открыв страницу на удаленном репозитории через веб-интерфейс. Обновление README.md на удаленном репозитории показано на рисунке 46.

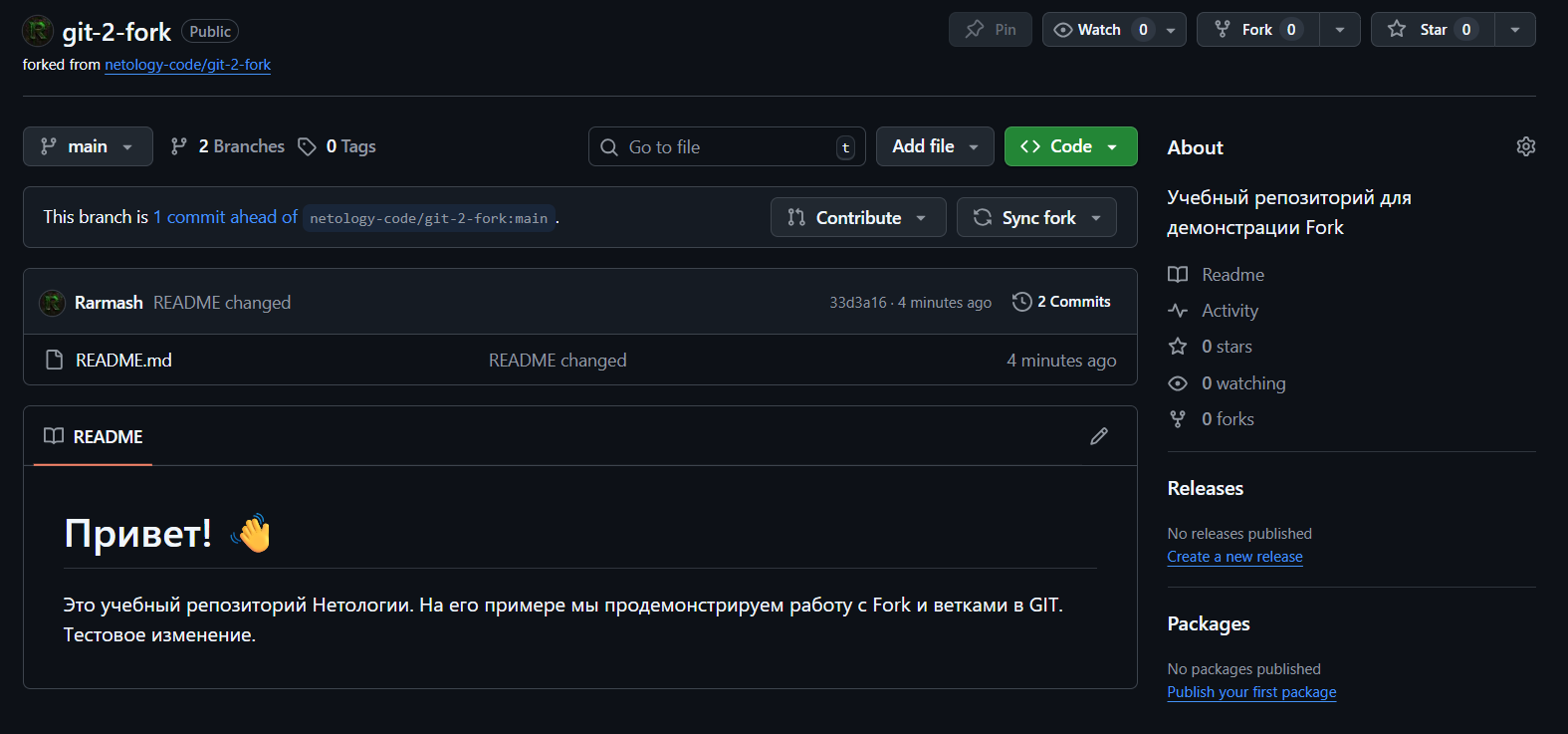


Рисунок 46 – Обновление README.md на удаленном репозитории

Создадим папку с именем conflict. Откроем для папки conflict терминал и инициализируем репозиторий через команду git init (рисунок 47). Откроем папку в Visual Studio Code, создайте в ней файл README.md. Добавим в файл заголовок «# Три факта обо мне». Ниже через пустую строку добавим список из трёх произвольных пунктов о себе (рисунок 48).



Рисунок 47 – Создание нового репозитория conflict

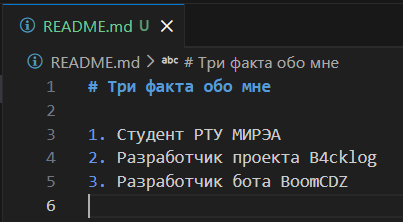


Рисунок 48 – README.md в новом репозитории conflict

Добавим все файлы в отслеживаемые с помощью команды git add и флага -A. Выполним коммит с соответствующим сообщением о первом коммите, после чего создадим новую ветку и сразу же на нее переключимся с помощью команду git checkout -b fix (рисунок 49).

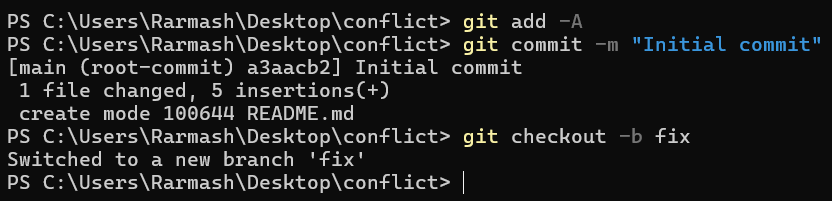


Рисунок 49 – Коммит измененных файлов и переключение на новую ветку

После переключения на новую ветку внесем изменения в README.md, после чего добавим его в отслеживаемые и выполним коммит с соответствующим сообщением об изменении README.md (рисунок 50).

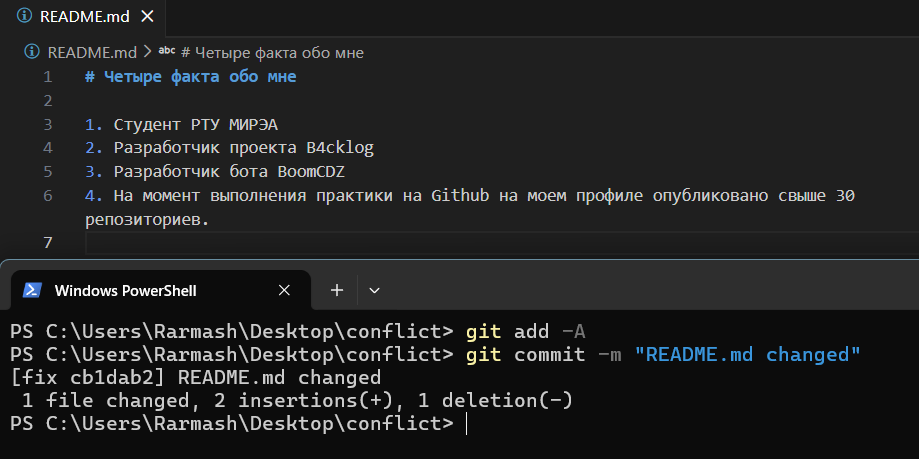


Рисунок 50 – Внесение изменений в файл README.md и выполнение коммита

После выполнения коммита вернемся на основную ветку и внесем изменения в файл README.md, похожие на те, что мы внесли в ветку fix, но поменяем четвертый факт на другой, после чего снова выполним коммит (рисунок 51).

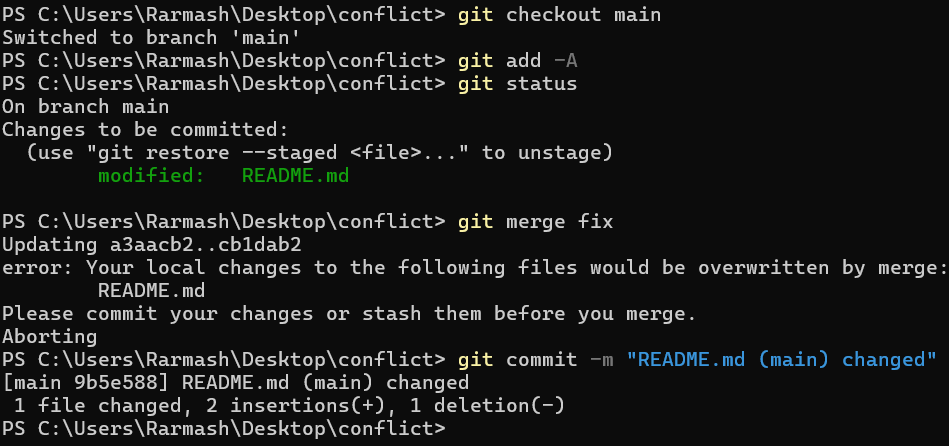


Рисунок 51 – Внесение изменений в файл README.md и выполнение коммита но на ветке main

Теперь попробуем слить изменения с помощью команды git merge. Git сообщит о конфликте (рисунок 52).

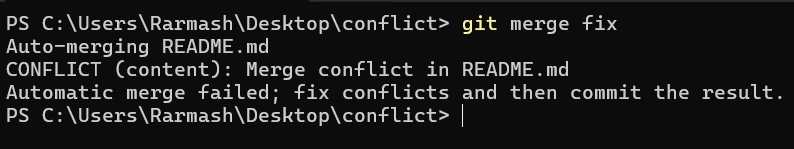


Рисунок 52 – Конфликт при попытке слияния веток

Посмотрим строку со словом CONFLICT. В ней написано, в каком именно файле произошёл конфликт. Откроем Visual Studio Code с файлом README.md и посмотрим, как Git показывает этот конфликт. Git всегда визуально показывает две версии изменений. Нужно самостоятельно принять решение: что удалить, а что оставить. С <<<<<<< HEAD и до ======= будет первая версия изменений в ветке, в которую мы сливаем изменения. От ======= до >>>>>>> fix будет вторая версия изменений из ветки, которую мы сливаем. Можно удалить одну или вторую версию. При этом обязательно нужно удалять технические символы <<<<<<< HEAD, ======= и >>>>>>> fix. Конфликт изображен на рисунке 53.

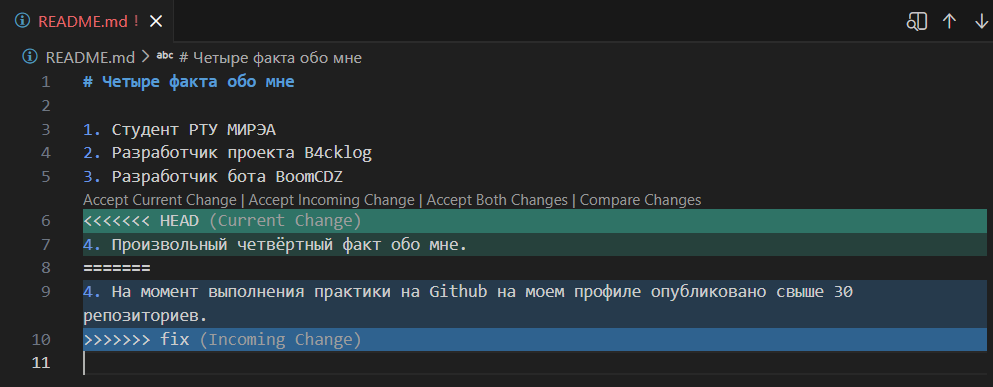


Рисунок 53 – Конфликт при попытке слияния веток

Внесем изменения и выполним коммит изменений (рисунок 51).

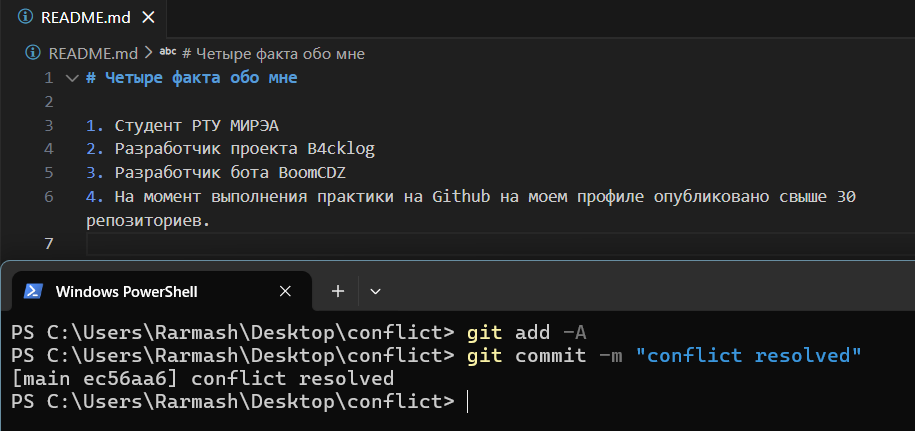


Рисунок 54 – Разрешение конфликта

Командная работа в Git и GitHub – это совместная реализация проектов несколькими разработчиками, при которой Git используется в качестве системы контроля версий локальных хранилищ отдельных участников команд, а GitHub – как общее удалённое хранилище для интеграции кода и коллективной работы над ним. Это позволяет командам работать параллельно над разными задачами в проекте, использовать ветки для изоляции изменений и последующего корректного слияния.

При командной разработке важно уметь:

* работать с коммитами;
* отправлять изменения из локального хранилища в удалённый репозиторий проекта (обновлять проект);
* синхронизировать локальные и удалённый репозитории;
* настраивать видимость файлов и папок репозитория для СКВ;
* создавать задачи (issues);
* отправлять изменения в удаленный репозиторий через pull request, который предполагает обсуждение и код-ревью.

При командной реализации проекта каждый разработчик создаёт для себя локальную копию удалённого репозитория (клонирование). После внесения изменений в локальную копию (коммит) обновляют общее удалённое хранилище.

В свою очередь, при обновлении удалённого общего хранилища из одной локальной папки, эти изменения необходимо внести во все остальные локальные хранилища (синхронизировать). Проиллюстрируем это на примере.

Создадим новый репозиторий colab-gav (рисунок 55).



Рисунок 55 – Создание нового репозитория

В консоли Git пропишем команду для клонирования репозитория при помощи ssh в определенную папку. Клонирование проекта показано на рисунке 56.



Рисунок 56 – Клонирование репозитория в определенную папку

Откроем в редакторе Visual Studio Code папку colab-first и создадим в ней файл README.md. Добавим в файл заголовок “# Командная работа – Rarmash”. После этого добавим файл в отслеживаемые, сделаем коммит с соответствующим сообщением о первом коммите и запушим проект на удаленный репозиторий. Выполненные действия показаны на рисунке 57.

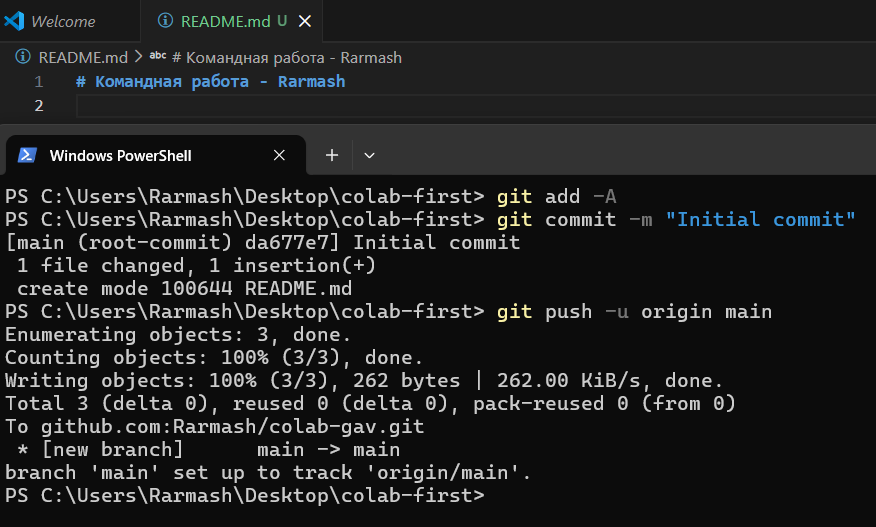


Рисунок 57 – Внесение изменений и их сохранение в репозитории

Проверим успешность сохранения изменений, зайдя на страницу GitHub. Результат показан на рисунке 58.

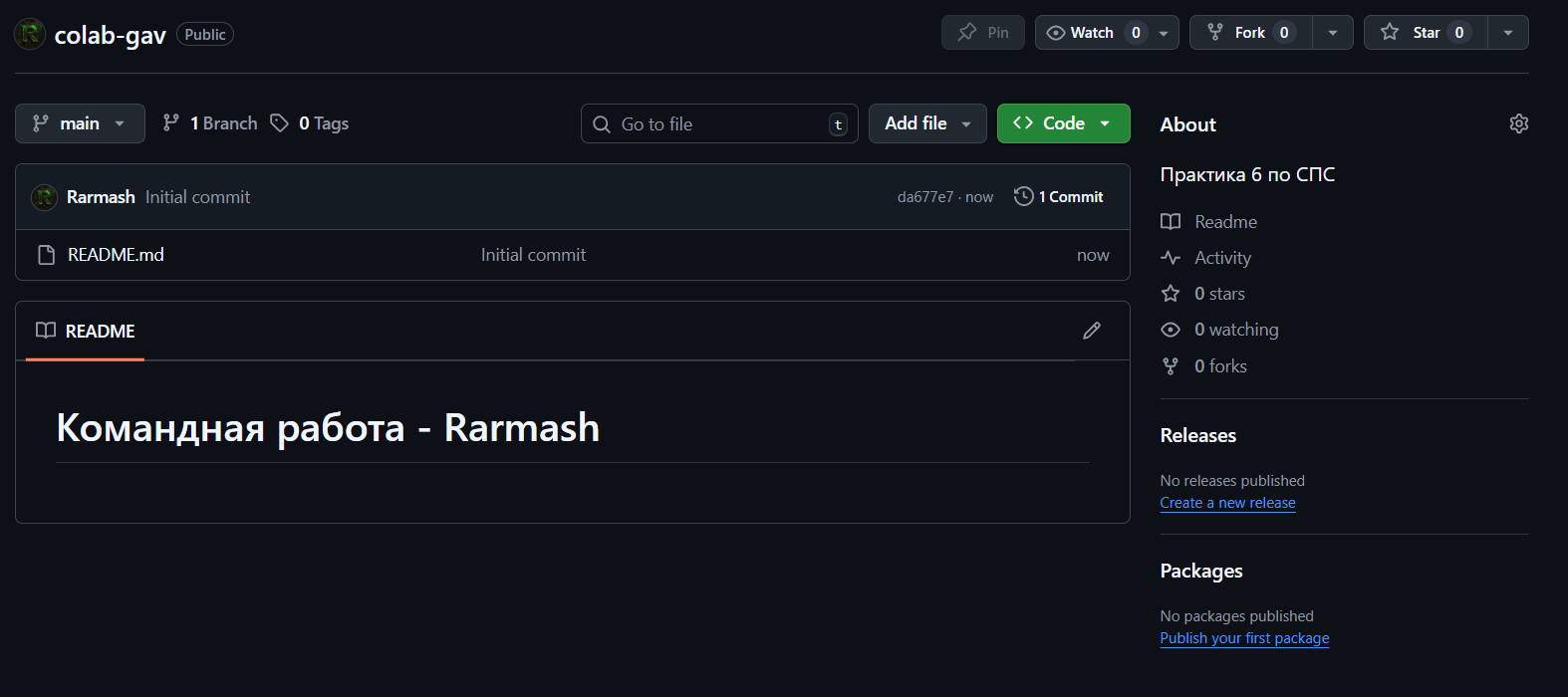


Рисунок 58 – Успешное внесение изменений в удаленном репозитории

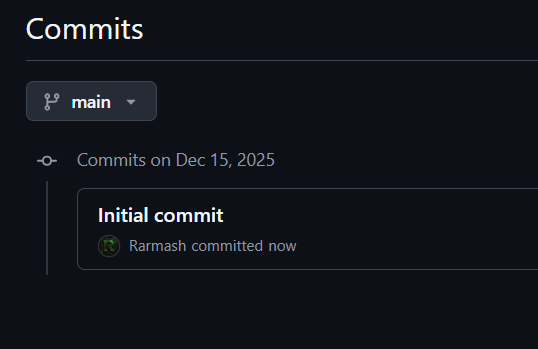


Рисунок 59 – Проверка наличия коммита

Вернемся в терминале в папку рабочего стола и снова выполним команду клонирования (рисунок 60).

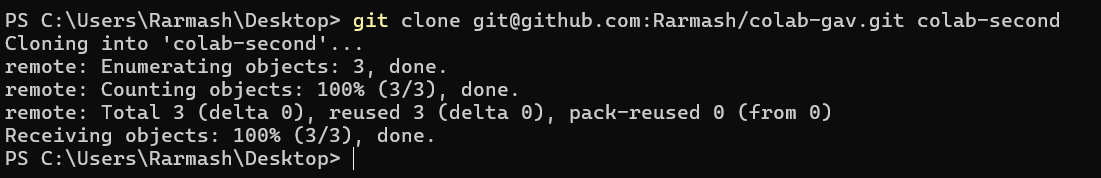


Рисунок 60 – Клонирование репозитория

Теперь здесь есть два локальных хранилища, связанные с удалённым репозиторием на GitHub. Откроем в редакторе Visual Studio Code вторую папку colab-second. Представим, что это наш коллега по группе разработки программного проекта. Создадим файл с именем sample.md и добавим в него заголовок “## Тестовый файл”. Добавим файл в отслеживаемые и закоммитим его. Отправим коммит в репозиторий с подписью «sample added». Выполненные действия показаны на рисунке 61.

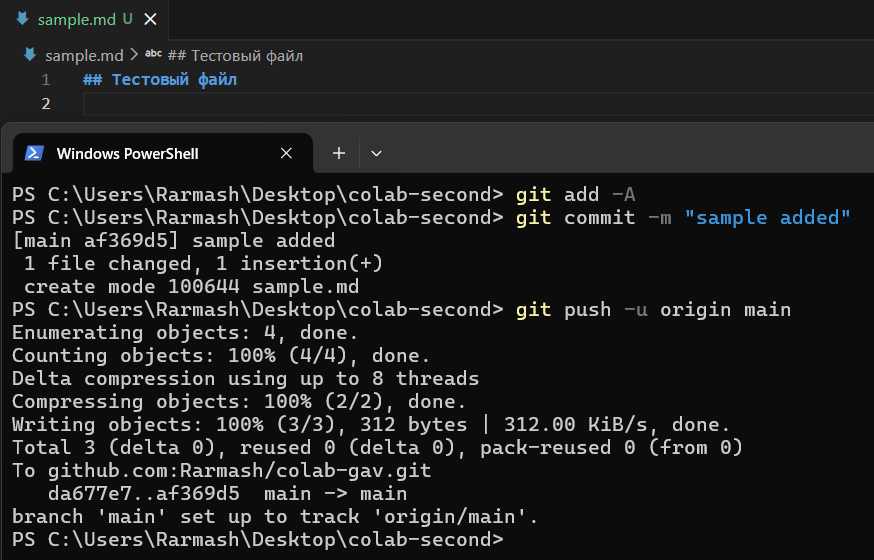


Рисунок 61 – Создание нового файла и выгрузка изменений в удаленные репозиторий

Вернемся к папке colab-first. Откроем её в редакторе Visual Studio Code и перейдем в неё в терминале. В редакторе в файл README.md добавим ниже заголовка через пустую строку текущее время и дату. Закоммитим изменения и попытаемся отправить их в удалённый репозиторий (рисунок 62).

В этот момент возникает ситуация, что локальный репозиторий не совпадает с удалённым. В удалённом репозитории уже есть коммит, который отправили из папки colab-second и которого нет в colab-first. Поэтому сначала каким-то образом необходимо обновить проект в папке colab-first.

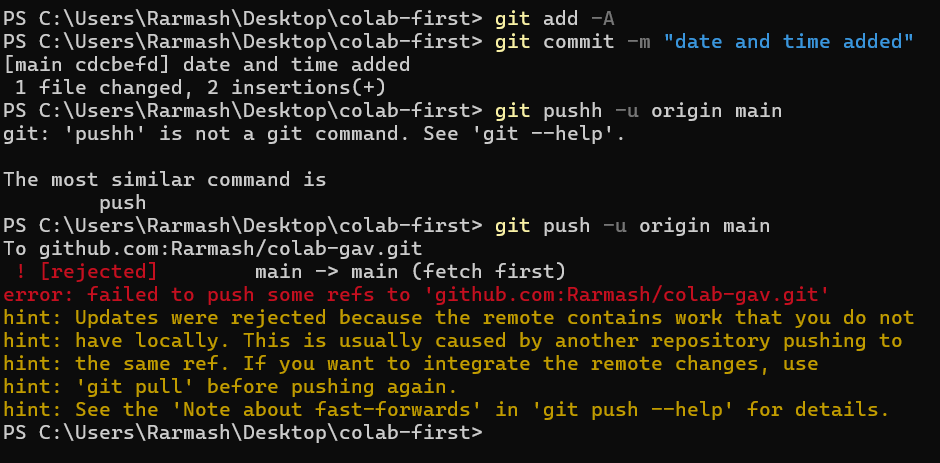


Рисунок 62 – Создание нового файла и выгрузка изменений в удаленные репозиторий

Чтобы забрать изменения из удалённого репозитория и обновить локальную копию, нужно использовать команду git pull. Выполним команду git pull (рисунок 63).

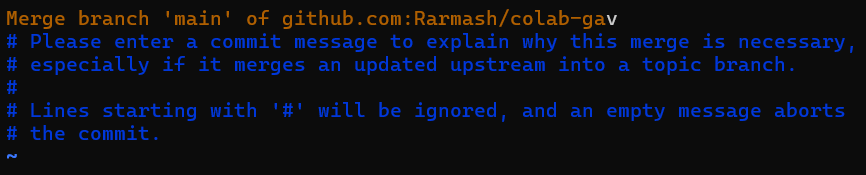


Рисунок 63 – Выполнение команды git pull

Поскольку «под капотом» этой команды скрываются git fetch и git merge, то на последнем этапе произойдёт слияние изменений в текущую историю проекта. В открывшемся окне редактора Visual Studio Code (исполнение команды git pull в терминале приостановится) можно ввести подпись для коммита слияния. Но часто оставляют стандартную подпись. Поэтому просто сохраним файл и закроем его (в терминале при этом завершится исполнение команды git pull). Результат действий показан на рисунке 64.

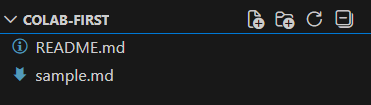


Рисунок 64 – Успешное слияние веток

Закончим то, из-за чего пришлось синхронизировать хранилища проекта – повторно отправим коммит из colab-first в удалённый репозиторий (рисунок 65). Откроем GitHub и убедимся, что все коммиты видны в репозитории (рисунок 66).

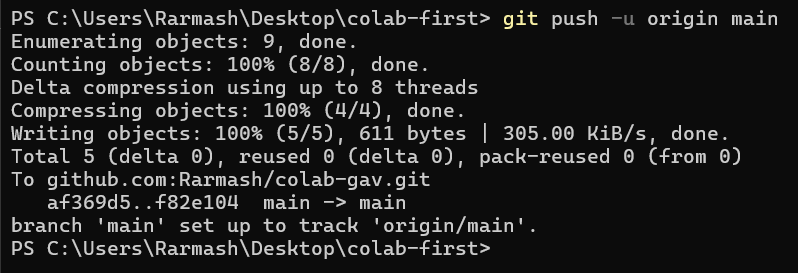


Рисунок 65 – Успешное выполнение пуша на удаленный репозиторий

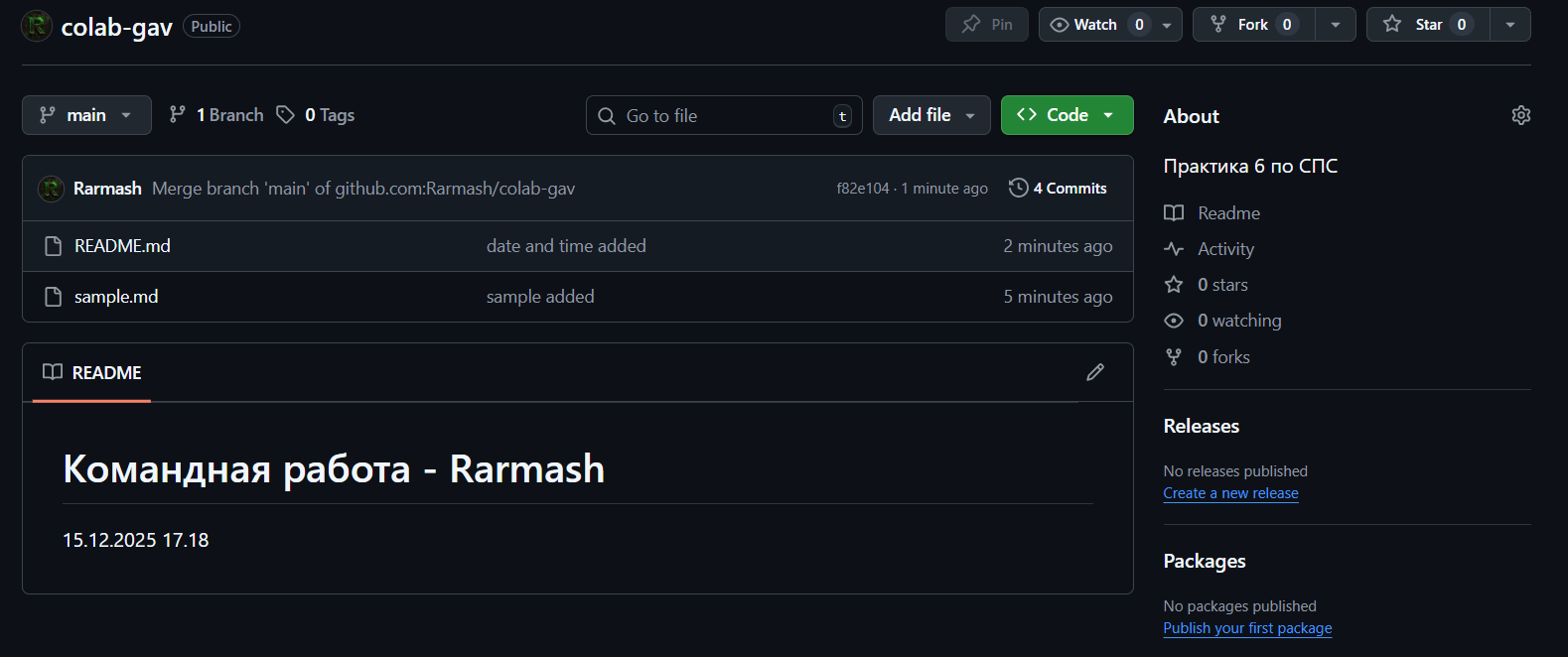


Рисунок 66 – Коммиты видны в репозитории

На этапе синхронизации могут возникать конфликты в файлах. Разрешите их, внимательно изучив комментарии, которые даёт Git. Таким образом при работе в команде всегда в начале работы и перед отправкой изменений в удалённое хранилище необходимо выполнять команду git pull.

Откроем терминал для папки colab-first. Выполним команду git log --oneline. Скопируем хеш самого первого коммита «Initial commit». Git выдаст довольно большую справку с предупреждением. Нас интересует только первая строка: Note: switching to ‘da677e7’. Выполненные действия показаны на рисунке 67.

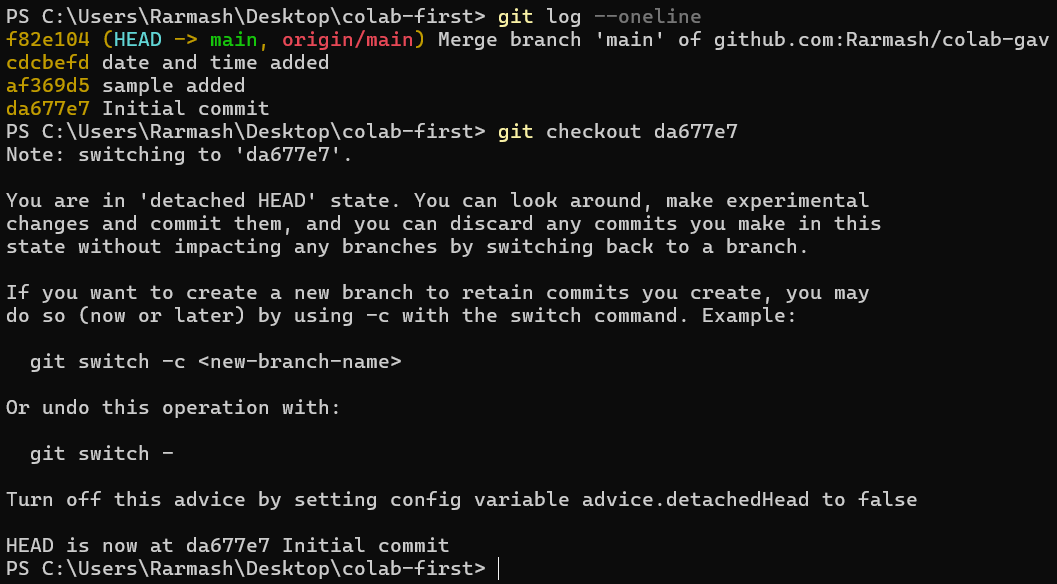


Рисунок 67 – Переход к первому коммиту

Это значит, что переключение на соответствующее состояние проекта произошло. Снова выполним команду git log --oneline. Таким образом мы вернулись назад к моменту, когда в проекте был только первый коммит, более поздних будто не существует. В редакторе Visual Studio Code убедимся, что в проекте снова только файл README.md с одним заголовком (без внесённых позже даты и времени). Изменения в файлах показаны на рисунке 68.

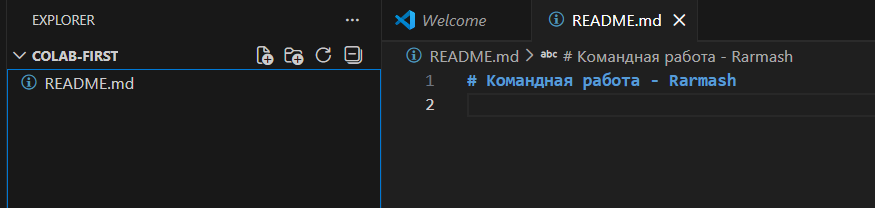


Рисунок 68 – Изменения в visual studio code

Чтобы посмотреть всю историю проекта в терминале, выполним команду “git log --oneline --all” (рисунок 69).

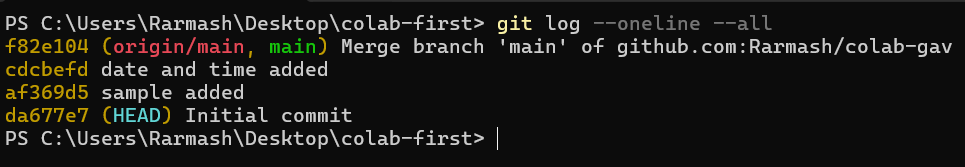


Рисунок 69 – Выполнение команды git log

Флаг --all требует от Git показать всю существующую историю. Тот коммит, на котором находимся сейчас, отмечен словом HEAD в скобках в начале строки. Вернёмся в реальность проекта, выполнив команду git checkout main (рисунок 70).

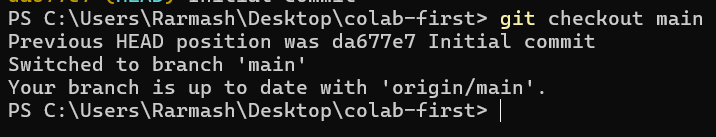


Рисунок 70 – Возврат к последнему коммиту проекта

Вместо main можно указать имя другой ветки, в которой мы сейчас находимся. Снова проверим историю: теперь HEAD написано возле последнего коммита. Мы вернулись к текущему состоянию проекта. В папке проекта снова два файла (рисунок 71).

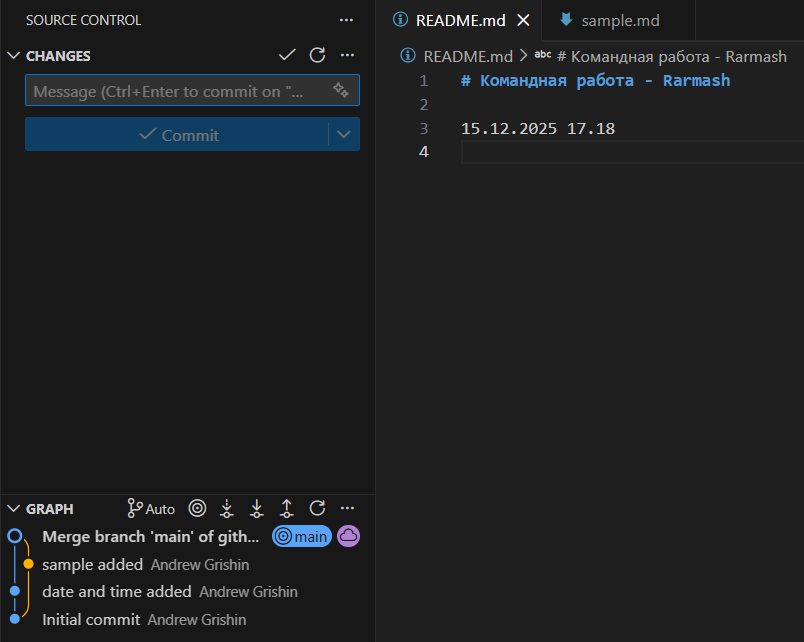


Рисунок 71 – Возврат к последнему коммиту проекта

Для отмены изменений (часто ошибочных), которые были внесены с последним коммитом, используется команда git revert. Посмотрите историю и скопируйте хеш последнего коммита. Выполните команду “git revert -m af369d5”. Результат выполнения команды показан на рисунке 72.

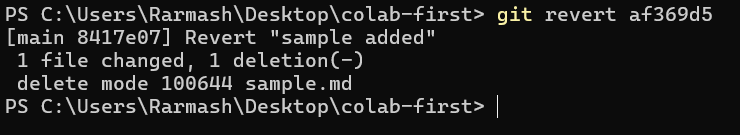


Рисунок 72 – Выполнение команды git revert

Открылось окно редактора, в котором нужно ввести подпись для обратного коммита. Оставим ту, что предложена по умолчанию. После закрытия редактора команда будет выполнена. При просмотре истории увидим, что был создан новый коммит. Отправим новый коммит в удалённый репозиторий (git push). Теперь на GitHub посмотрим, что произошло (рисунок 73).



Рисунок 73 – История коммитов на GitHub

Сначала в истории найдите и откройте коммит «date and time added». В нём была добавлена строка под заголовком с датой и временем (рисунок 74).



Рисунок 74 – Коммит с изменением README.md

Теперь откройте последний коммит “Revert "sample added"”. В нём добавление файла sample.md было отменено (рисунок 75). При отмене коммитов могут возникать конфликты. Разрешайте их, как и в любых других случаях.

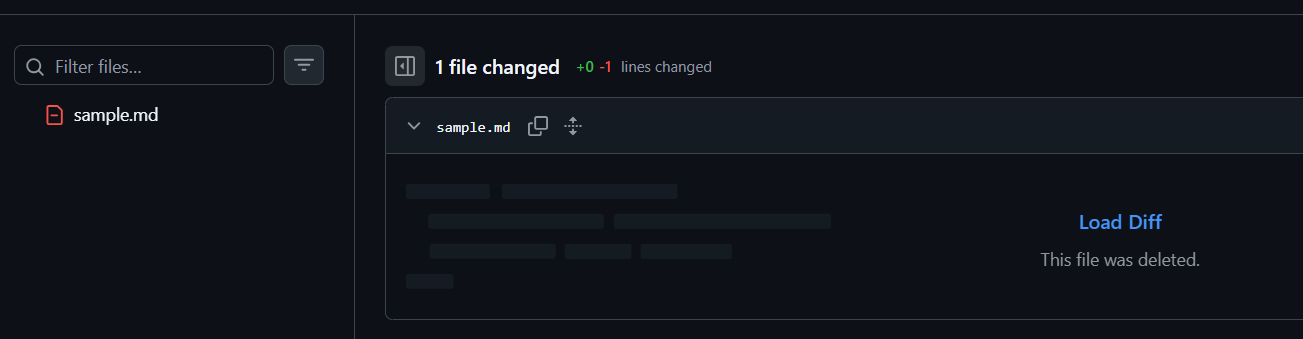


Рисунок 75 – Коммит с sample.md

Файл .gitignore – это файл-настройка, который создаётся в папке проекта. В нём указывается, какие файлы нужно игнорировать при публикации на GitHub. При этом:

* имя файла должно начинаться с точки, а в конце не должно быть никакого иного расширения;
* правила игнорирования сработают только для файлов, не добавленных в индекс, неотслеживаемых, т.е. до команды git add.

Создадим в папке проекта colab-first папку private и внутри неё файл today.md. В файле напишем список учебных занятий на сегодня. Выполним команду git status и убедимся, что Git видит папку private (рисунок 76).

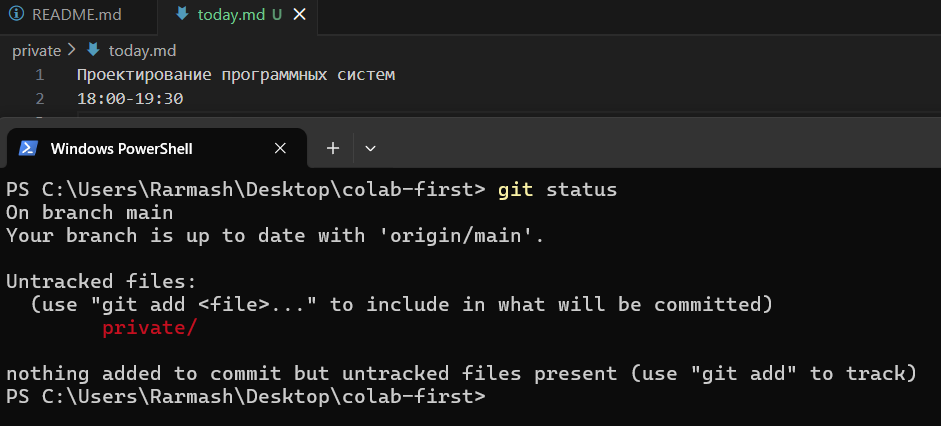


Рисунок 76 – Содержание файла today.md и выполнение команды git status

В папке colab-first создадим файл .gitignore. В этом файле напишем “private/” обязательно со слэшем в конце, т.к. игнорировать нужно папку, а не файл с таким именем. Содержание файла показано на рисунке 77.



Рисунок 77 – Содержание файла .gitignore

Снова просмотрим статус – папка private больше не видна Git. Новый файл .gitignore закоммиттим и отправим в репозиторий, чтобы все коллеги могли пользоваться правилами игнорировать и дописывать в этот файл свои данные (рисунок 78).



Рисунок 78 – Git более не видит private и отправка коммита с .gitignore

Создадим новый репозиторий на Github, назовем его с использованием своих инициалов (рисунок 79). Клонируем репозиторий на ПК. В локальной папке проекта создадим краткое описание в файле README.md со своей фамилией и текущей датой и временем. Описанные действия показаны на рисунке 80.

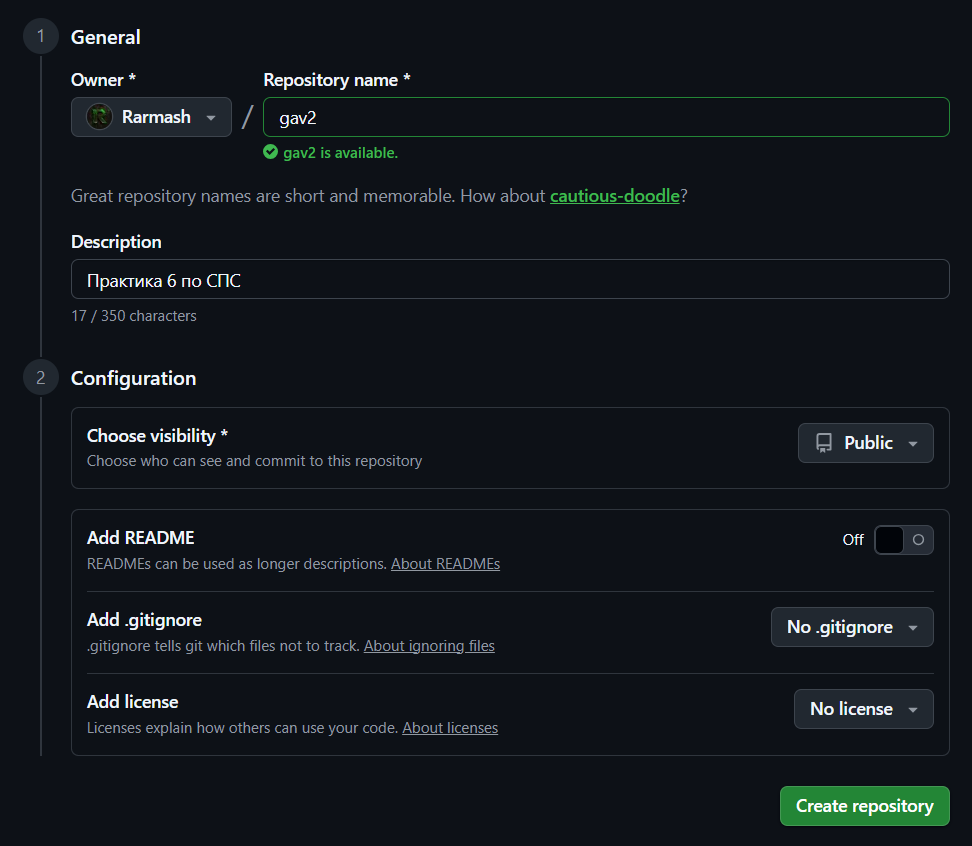


Рисунок 79 – Создание нового GitHub репозитория

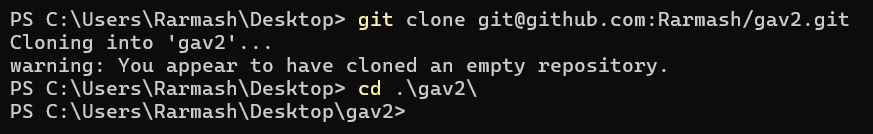


Рисунок 80 – Клонирование и переход в созданный репозиторий

Создадим проект на Python. Будем использовать изолированную среду. С помощью Virtualenv создадим её в папке проекта. Т.к. она создана в папке проекта, следует исключить её из управления VCS. Для этого в корне репозитория создадим файл .gitignore с необходимым паттерном. Указанные действия показаны на рисунке 81.

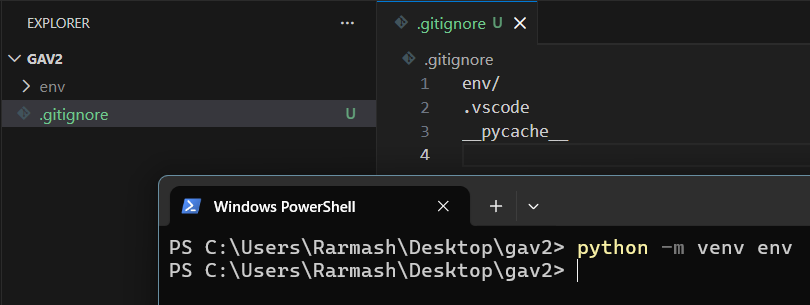


Рисунок 81 – Содержание файла .gitignore и создание виртуального окружения

В корне репозитория создайте файл hypotenuse.py, показанный на рисунке 82.

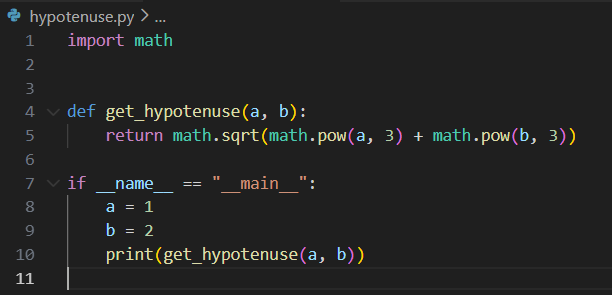


Рисунок 82 – Содержание файла hypotenuse.py

Создадим первый коммит проекта, т.е. зафиксируем изменения в репозитории с осмысленным сообщением. Коммит изображен на рисунке 83.

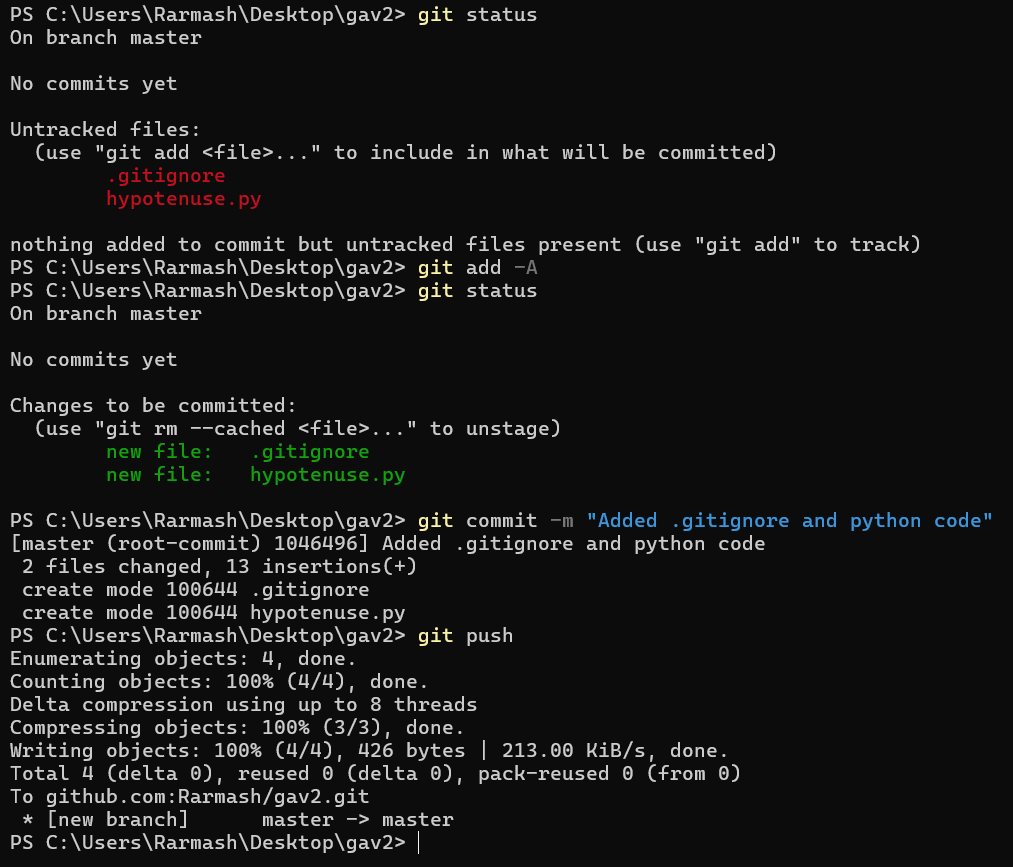


Рисунок 83 – Создание первого коммита проекта

Изменим исходный код так, чтобы программа через стандартный поток ввода запрашивала ввод чисел a и b, при этом в стандартный поток вывода поступал результат. Результат изменения программы показан на рисунке 84.

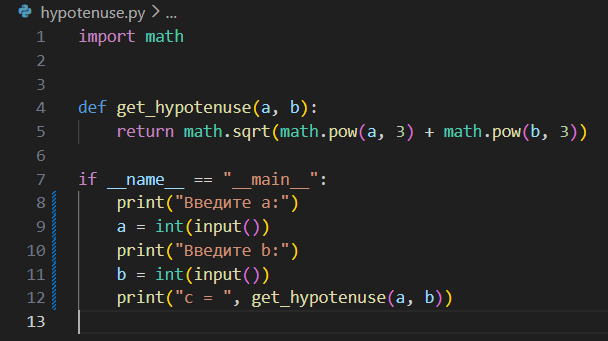


Рисунок 84 – Измененное содержание файла hypotenuse.py

Зафиксируем новую версию программы и отправим изменения в удалённый репозиторий (рисунок 85).

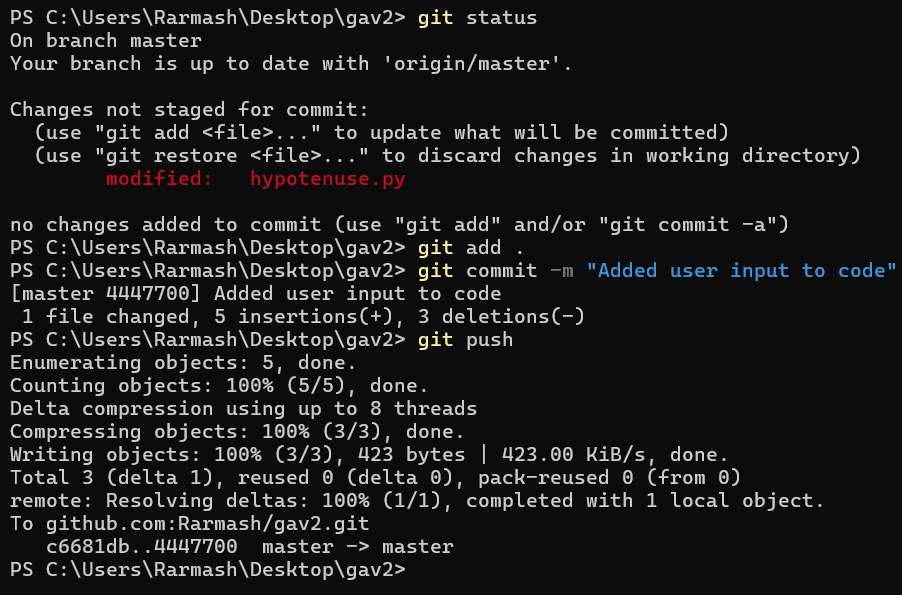


Рисунок 85 – Фиксация файла hypotenuse.py и отправка коммита на удаленный репозиторий

Проверим, что история коммитов доступна в удаленном репозитории. Страница репозитория указана на рисунке 86.

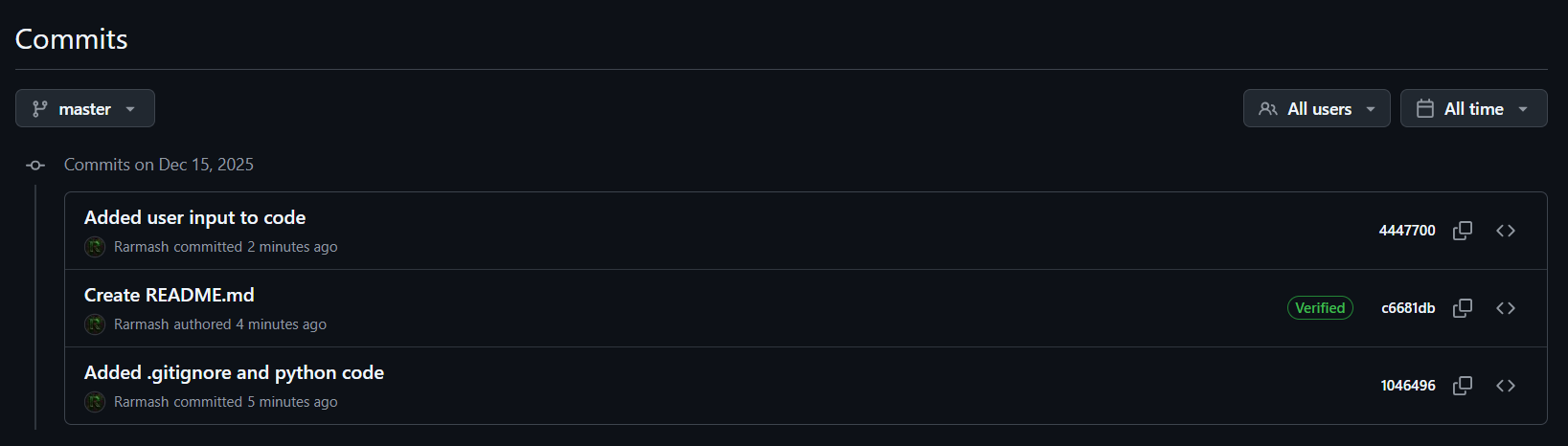


Рисунок 86 – История коммитов

В локальной копии репозитория создадим новую локальную ветку feature1 (рисунок 87). Внесем изменения в ветке feature1. Например, добавим расчет площади прямоугольного треугольника, при этом сделаем в этой функциональности осознанную ошибку. Код на Python показан на рисунке 88.

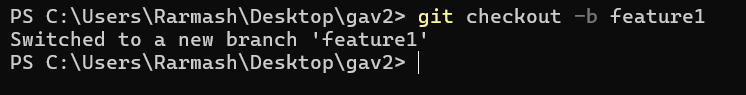


Рисунок 87 – Создание и переход на новую ветку feature1

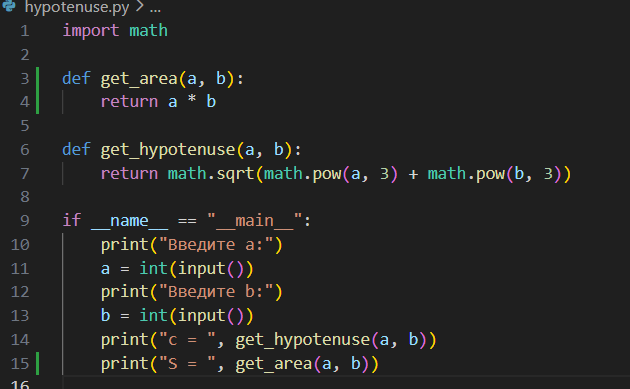


Рисунок 88 – Код программы с ошибкой в функции вычисления площади

Зафиксируем и отправим изменения в удалённый репозиторий (рисунок

89).

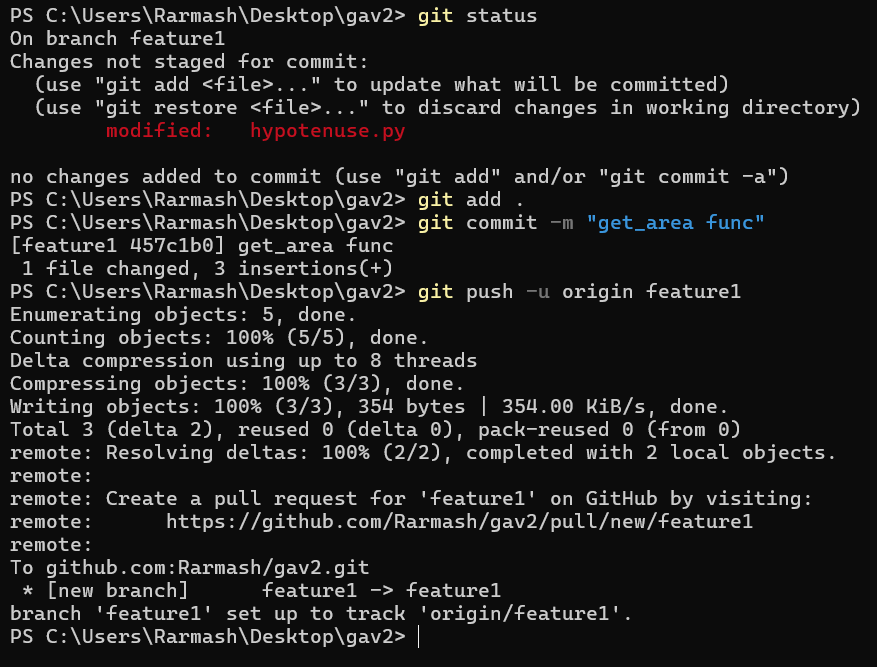


Рисунок 89 – Отправка изменений в удаленный репозиторий

Переименуем исходный файл в calculator.py, зафиксируем и отправим изменения в удалённый репозиторий. Процесс показан на рисунке 90.

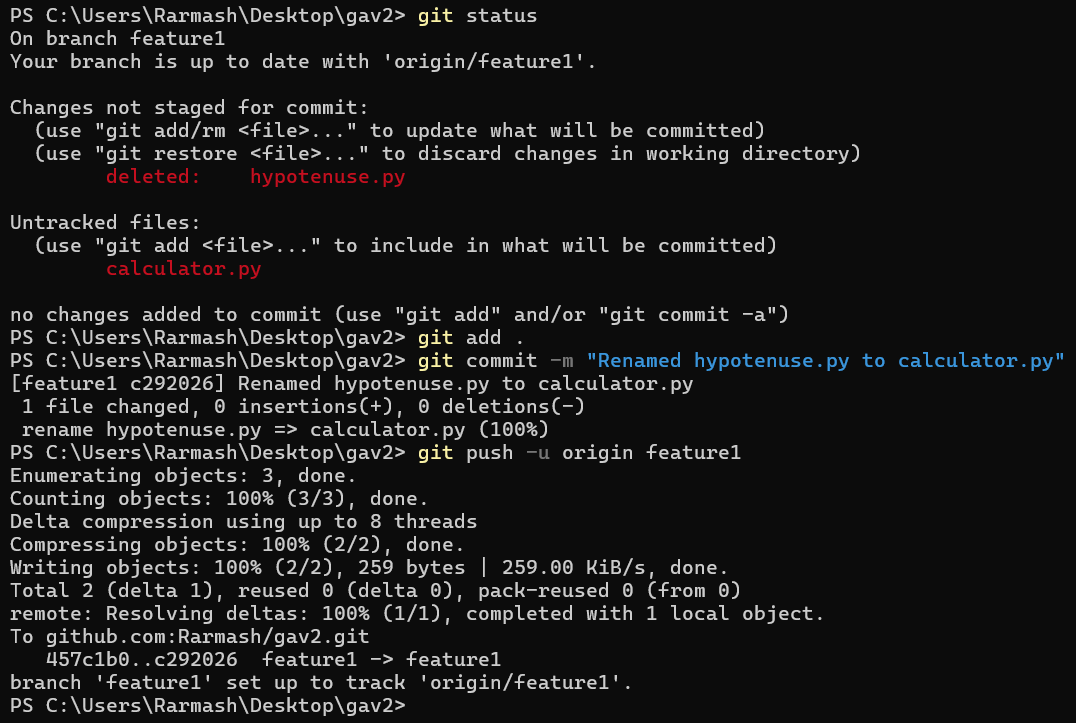


Рисунок 90 – Переименование файла и отправка изменений в удаленный репозиторий

Проверим, что ветка feature1 доступна в удаленном репозитории. Страница с кодом в ветке feature1 показана на рисунке 91.

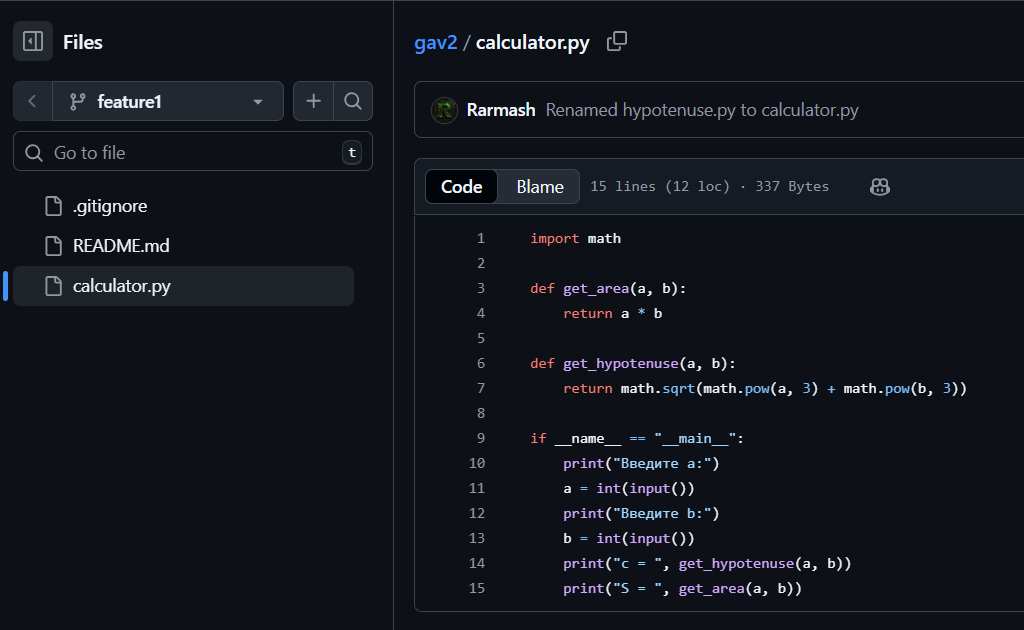


Рисунок 91 – Код в ветке feature1 в удаленном репозитории

На Github начнем формировать запрос на слияние веток на странице репозитория или 1) по кнопке «Compare @ pull request», или 2) во вкладке «Pull requests». Выберем исходную (feature1) и целевую (main) ветки, сравним их, заполним поля и нажмем «Create pull request» (рисунок 92).

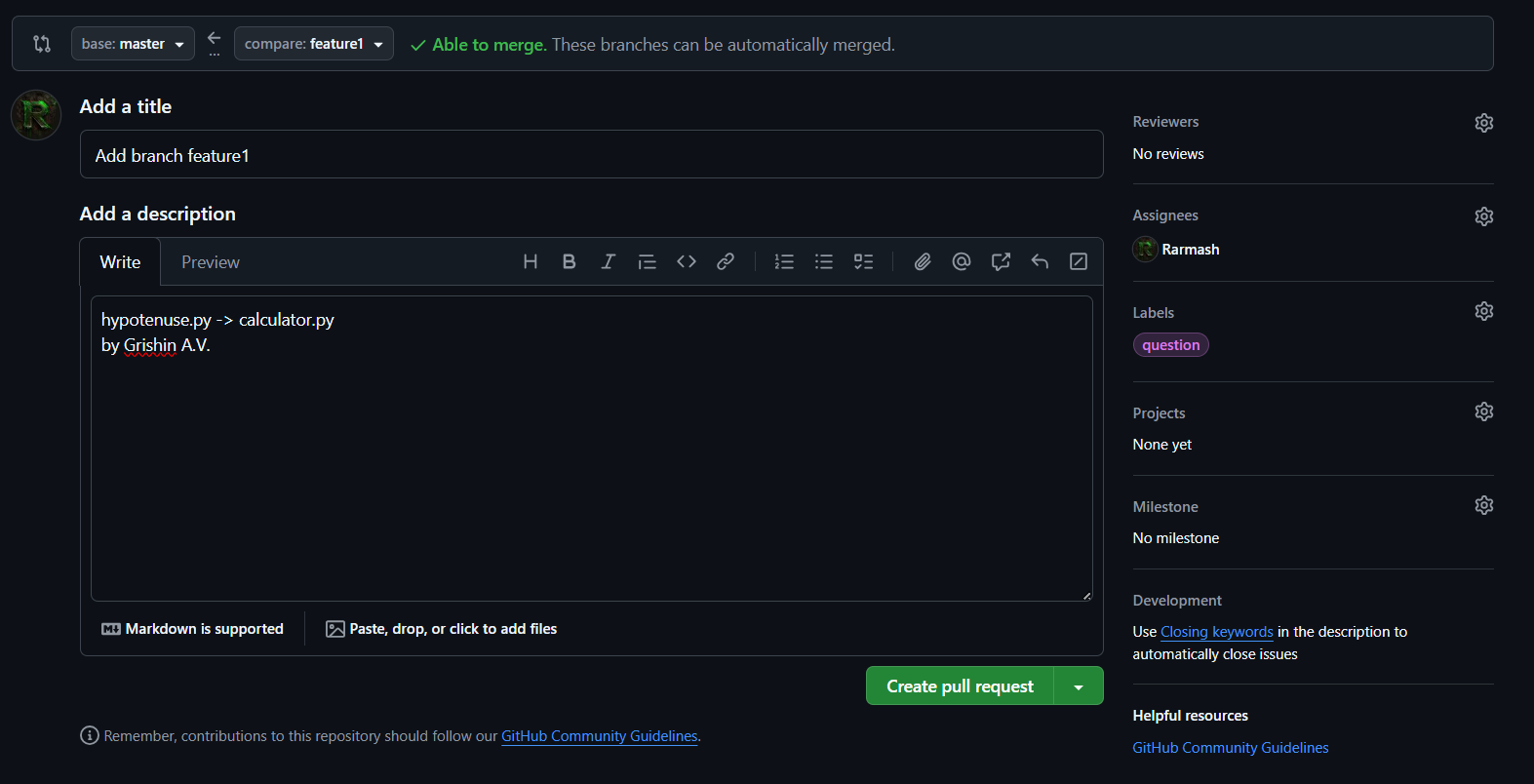


Рисунок 92 – Запрос на слияние веток в проекте

Осуществим проверку запроса на слияние. В случае отсутствия замечаний и проблем подтвердим запрос по кнопке «Merge pull request» и «Confirm merge», а затем «Delete branch» (рисунок 93). Убедимся, что запрос окажется во вкладке «Pull requests» → «Closed» (рисунок 94).

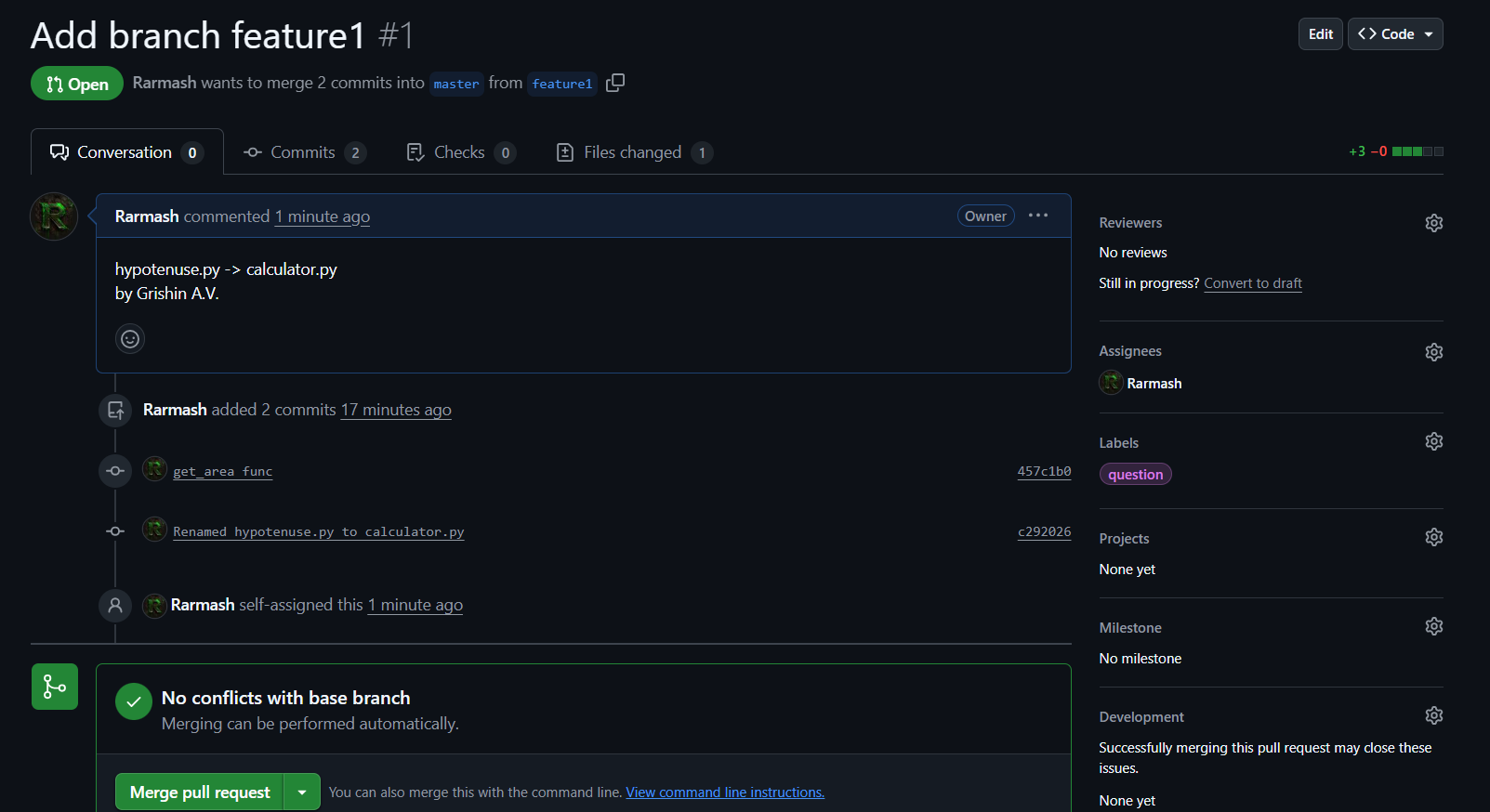


Рисунок 93 – Рассмотрение запроса на слияние веток в проекте

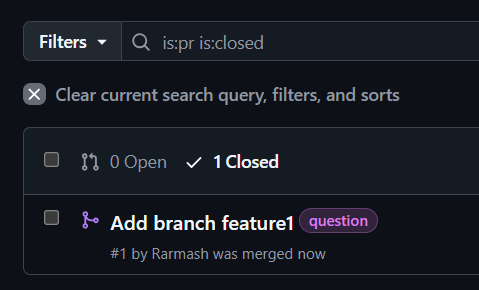


Рисунок 94 – Запрос переместился в закрытые

Скинем внесённые изменения с удалённого репозитория на локальное хранилище проекта. Удалим локальную ветку feature1. Эти этапы показаны на рисунке 95.

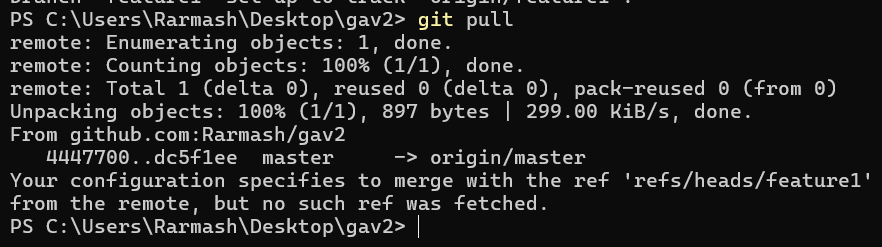


Рисунок 95 – Загрузка изменений и удаление ветки

Проверим в редакторе, что сохранился окончательный вариант calculator.py. Экран редактора показан на рисунке 96.

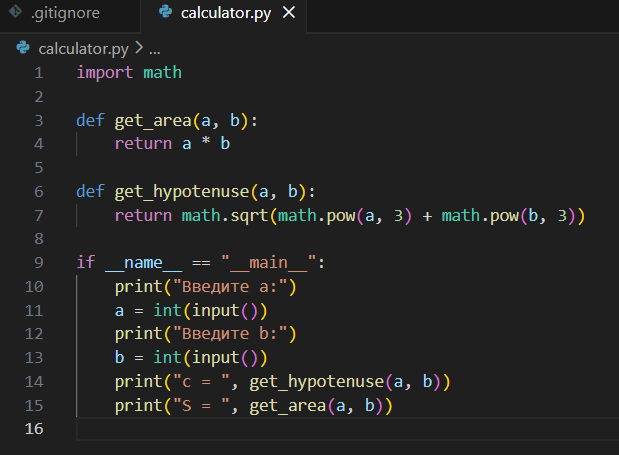


Рисунок 96 – Сохранившиеся изменения в редакторе

В локальной папке создайте новую ветку patch1. Создание новой ветки показано на рисунке 97.



Рисунок 97 – Создание новой ветки

В ней в файл calculator.py добавим комментарии, показанные на рисунке 98.

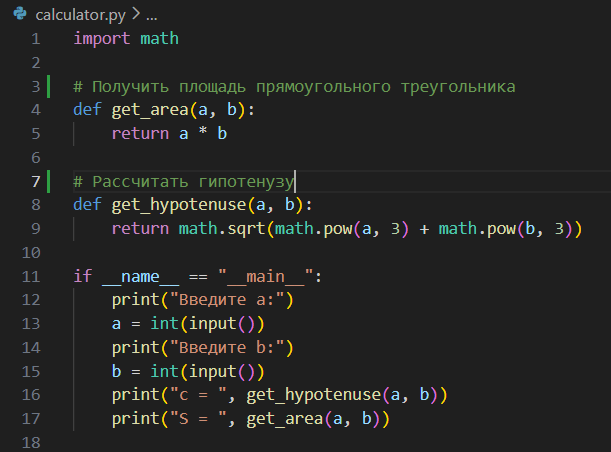


Рисунок 98 – Добавление комментариев к коду

Создадим коммит и отправим изменения в удалённый репозиторий. Процесс показан на рисунке 99.

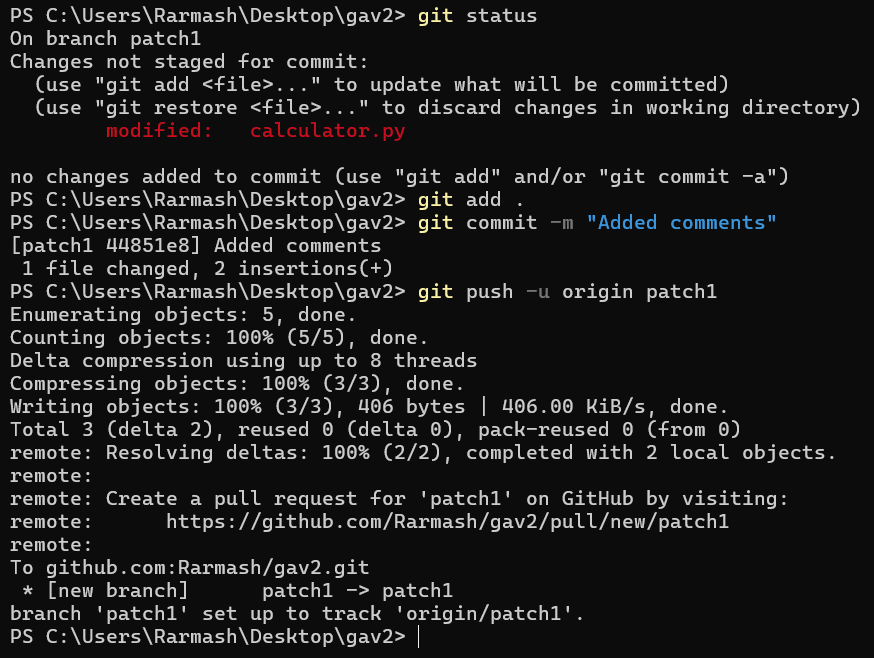


Рисунок 99 – Отправка изменений в удаленный репозиторий

Вернемся на ветку main и отредактируем код, поменяв местами две функции. Процесс перехода на ветку и изменение кода показаны на рисунке 100.

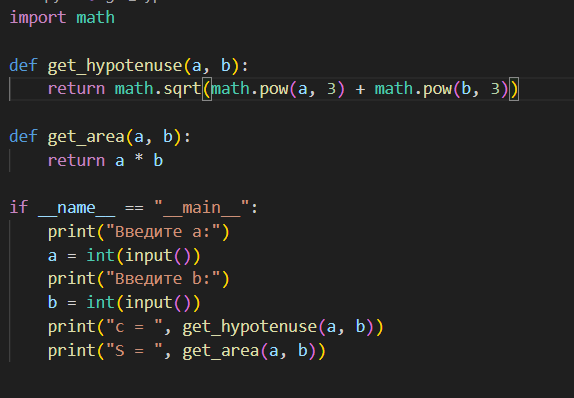


Рисунок 100 – Смена ветки и изменения в коде

Создадим коммит и отправим изменения на удаленный репозиторий.

Отправка изменений и создание коммита показаны на рисунке 101.

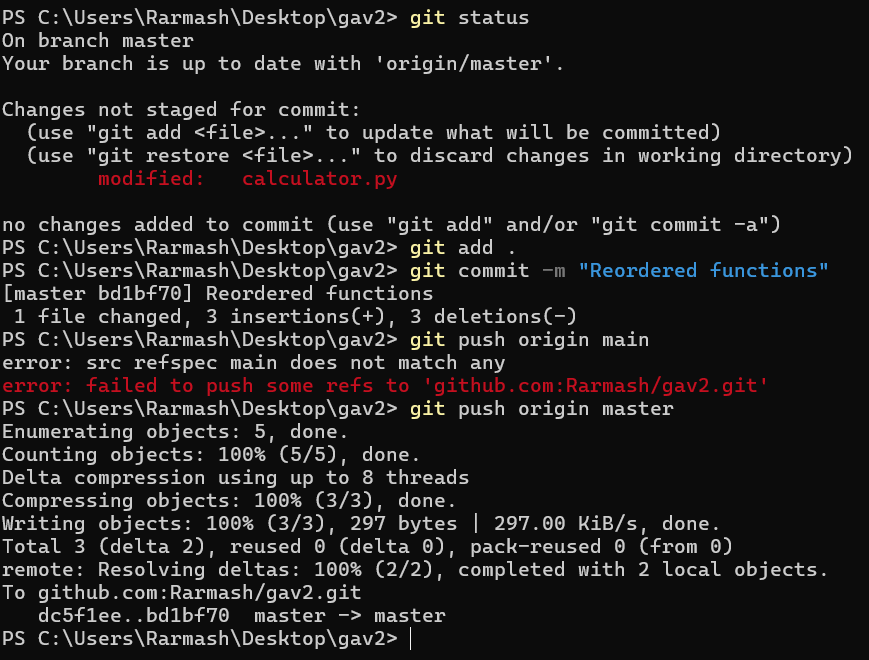


Рисунок 101 – Отправка изменений на удаленный репозиторий

Перейдем на GitHub и увидим, что автоматически решить конфликт невозможно. Сообщение о конфликте показано на рисунке 102.

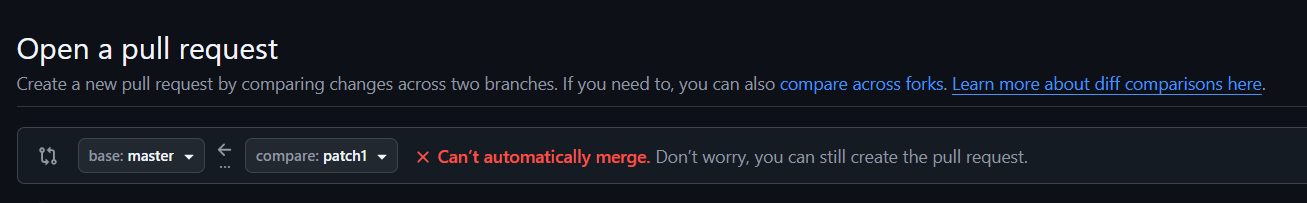


Рисунок 102 – Сообщение об ошибке слияния веток

Вернемся на ветку patch1 и начнем процесс слияния. Выполнение необходимых команд показано на рисунке 103.

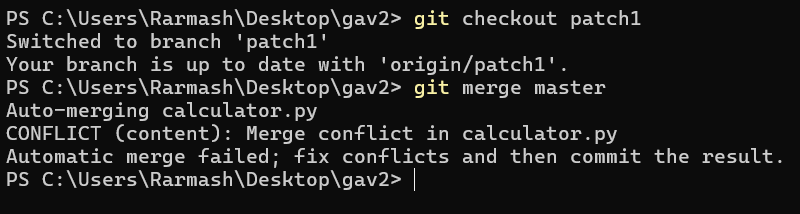


Рисунок 103 – Переход на ветку patch1 и начало процесса слияния

Откроем Visual Studio code и увидим конфликт, который необходимо решить, показанный на рисунке 104. Решение конфликта показано на рисунке 105.

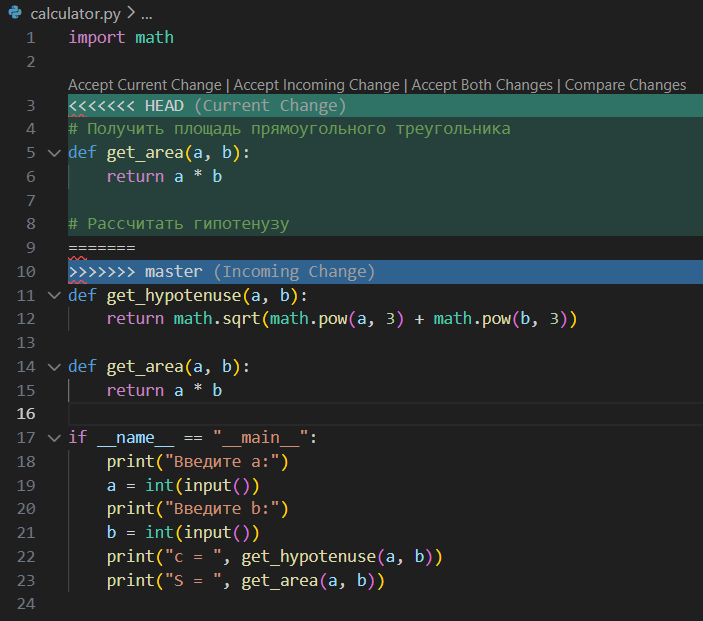


Рисунок 104 – Конфликт слияния

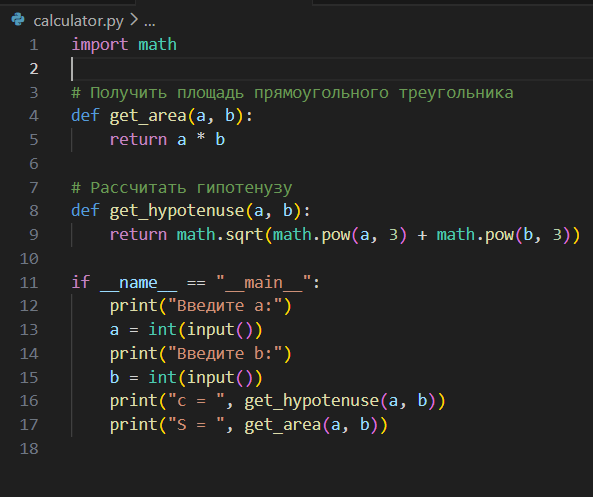


Рисунок 105 – Решение конфликта и итоговый файл

Проверим, что запрос на слияние закрыт. Закрытые запросы на слияние показаны на рисунке 106.



Рисунок 106 – Закрытые запросы на слияние

Создадим новую issue “Submit a report”. Опишем задачу и срок исполнения, упомянем свой аккаунт в качестве исполнителя, выберем метку “good first issue”. В день загрузки закроем эту задачу. Создание задачи показано на рисунке 107.

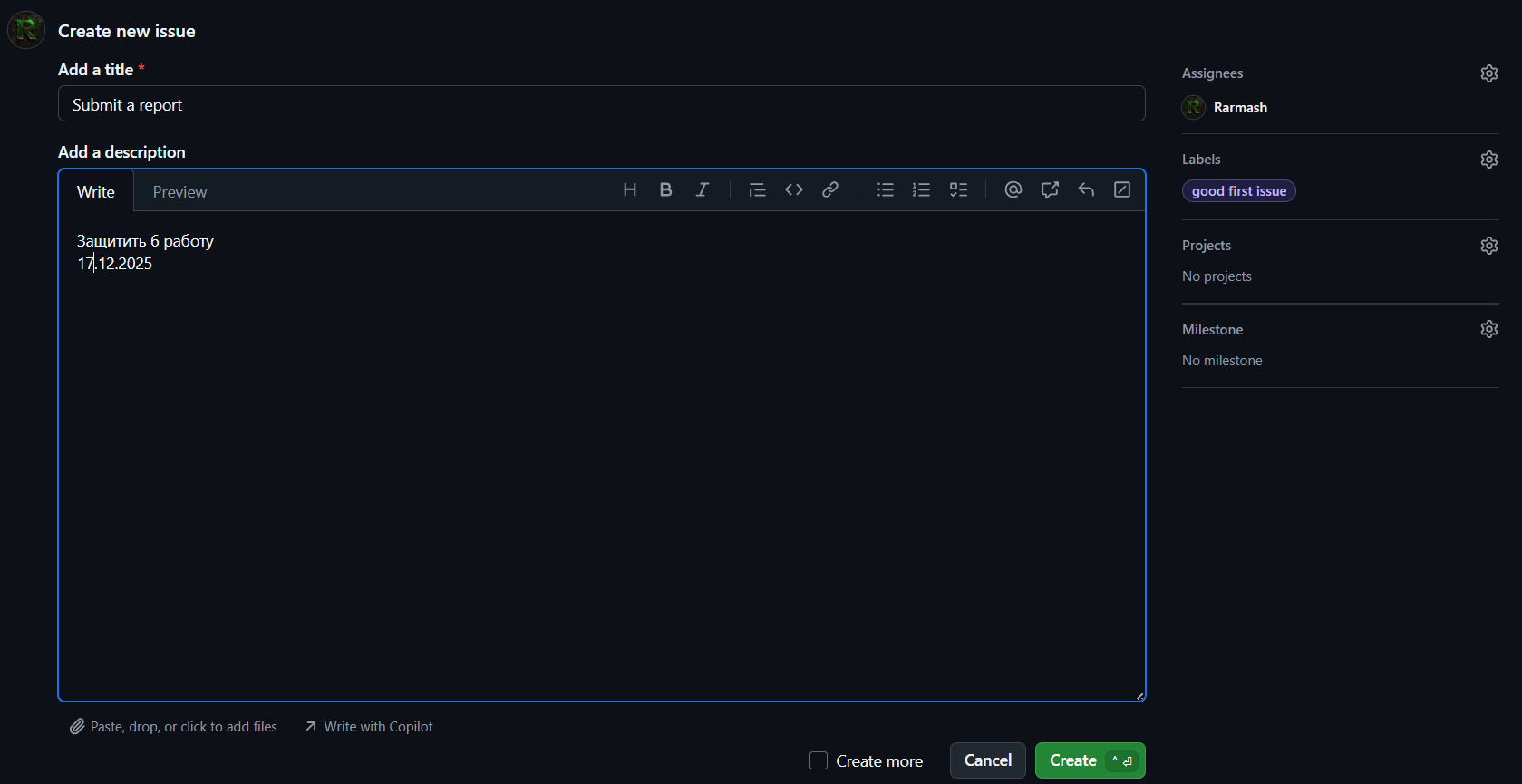


Рисунок 107 – Создание новой задачи

# Вывод

В ходе практической работы подробно изучены и практически отработаны ключевые аспекты системы контроля версий Git: создание и настройка репозиториев, работа с коммитами, ветками, слияниями, откатами и разрешением конфликтов. Освоены механизмы взаимодействия с удалённым хранилищем GitHub, включая SSH-аутентификацию, push/pull, клонирование, форки и Pull Request. Практически реализован небольшой Python-проект с использованием виртуального окружения, правильной структурой .gitignore, ветвлением по сценарию feature-branch, исправлением ошибок и обработкой конфликтов при слиянии. Отработаны инструменты командной разработки: синхронизация нескольких локальных копий, код-ревью через PR и планирование задач через Issues. Все этапы выполнены успешно, что подтверждает приобретение необходимых навыков по сопровождению программных систем, коллективной разработке и применению современных практик контроля версий..