

Technical Guide

Project Name: Gitulyse

Authors:

- Afolabi Fatogun - 20409054
- Adrian Irwin - 20415624

Supervisor: Stephen Blott

Finish Date: 19/04/2024

Abstract

The Gitulyse Technical Guide is a comprehensive resource for examiners to delve into Gitulyse's internal structure. Gitulyse is a tool designed to extract, process, and visualise data from the platform, Github, that facilitates collaborative Git (VCS) usage for projects hosted on the platform. It enables users to gain insights into their software development processes. This guide provides a detailed overview of Gitulyse's architecture, functionality, and usage, covering key aspects such as data extraction, performance optimisation, visualisation design, and collaboration workflows. Additionally, it also addresses common challenges encountered during Gitulyse's development. By leveraging the insights and practices outlined in this guide, examiners can effectively understand and utilise Gitulyse to enhance their software development workflows, streamline decision-making processes, and drive continuous improvement in their projects.

Table of Contents

Abstract.....	0
Table of Contents.....	1
Project Overview.....	4
Motivation.....	4
Research.....	6
Use Cases.....	7
Design.....	8
High-Level Design.....	8
Sequence Diagram.....	8
Data Flow Diagram.....	9
System Architecture.....	10
React Frontend:.....	10
Flask Backend:.....	10
Mongo Database:.....	11
Changes to the Original Architecture.....	11
Implementation.....	12
Bug Tracking.....	12
Continuous Integration.....	13
Defaults.....	13
Stages.....	13
Jobs.....	13
.standard-rules:.....	13
Frontend Lint.....	13
Backend Test.....	14
Conclusion.....	14
Features.....	16
Login.....	16
Logout.....	17
Repo Information Page.....	18
Modular Components.....	21
CodeContributions Component.....	21
IssueTracking Component.....	25
PullRequests Component.....	27
PercentagePullrequests Component.....	29
PercentageIssues Component.....	31
User Page.....	33
Frontend Implementation.....	33
Backend Implementation.....	33

Comparison Page.....	36
Backend Implementation.....	36
Frontend Implementation.....	36
Problems Resolved.....	38
Data Extraction and Processing Challenges:.....	38
Visualisation Design and Usability:.....	38
Results.....	39
Testing Strategy.....	39
Backend Integration Testing with Pytest (API Endpoints):.....	39
Frontend Unit Testing with Jest:.....	39
Ad-hoc Testing:.....	39
User Testing:.....	39
Frontend Unit Testing with Jest.....	40
Unit Test Coverage.....	40
Test Environment Setup.....	41
Test Cases.....	42
HomePage.test.js.....	42
nav.test.js.....	42
repoComponents.test.js.....	42
RepoPage.test.js.....	43
UserComparePage.test.js.....	43
userComponents.test.js.....	43
UserPage.test.js.....	43
Backend Integration Testing with Pytest (API Endpoints).....	43
Integration Test Coverage.....	43
Test Environment Setup.....	45
Test Cases.....	45
1. test_commits.py.....	45
2. test_db.py.....	45
3. test_factory.py.....	46
4. test_issues.py.....	46
5. test_pull_requests.py.....	46
6. test_repos.py.....	46
7. test_search.py.....	47
8. test_users.py.....	47
Ad-Hoc Testing.....	48
Login Function Testing.....	48
Home Page/ Nav Testing.....	49
User Page Testing.....	54
Compare Page Testing.....	55
Info Page Testing.....	56

User Testing.....	61
Future Work.....	65

Project Overview

Gitulyse is a comprehensive tool designed to analyse GitHub repositories and user data, providing insights and metrics essential for effective software development management. With its user-friendly interface and powerful features, Gitulyse offers a holistic view of project history, facilitating informed decision-making and optimisation of development workflows.

At its core, Gitulyse leverages advanced algorithms to parse through the extensive data within GitHub repositories, extracting valuable information such as commit patterns, code contributions, and project activity trends. Developers and project managers can utilise these insights to identify bottlenecks, assess team productivity, and track the evolution of codebase quality over time.

One of Gitulyse's notable capabilities is its ability to generate visualisations, including graphs and charts, that intuitively depict repository dynamics and collaboration patterns. These visual aids enhance comprehension and aid in communicating findings across teams.

Moreover, Gitulyse offers customisable reporting functionalities, allowing users to view summaries and metrics tailored to their needs. Whether it's monitoring individual developer contributions or evaluating the impact of code refactorings, Gitulyse provides the necessary tools for comprehensive repository analysis.

Overall, Gitulyse is a valuable asset for those seeking to optimise their software development processes, enabling them to harness the full potential of their Github repositories and drive continuous improvement in their projects.

Motivation

The motivation behind the development of Gitulyse stems from the growing complexity of software development projects and the need for tools that can provide deeper insights into Git repositories, as well as the need for people in management positions to monitor the progress of programmers under their management. As software projects become larger and more distributed, managing version control systems effectively becomes increasingly challenging. The traditional Github client offers basic functionality for analysis but lacks the comprehensive analysis capabilities needed to understand repository dynamics and optimise development workflows. Gitulyse addresses this gap by offering a holistic solution for repository analysis, empowering organisations to make informed decisions and drive continuous improvement in their software development processes.

Project management: Gitulyse facilitates project management by offering comprehensive insights and metrics that enable project managers to track progress, identify bottlenecks, optimise resource allocation, monitor compliance and security, foster collaboration, and drive continuous improvement in software development projects. Through its analysis capabilities, Gitulyse provides real-time visibility into code contributions, commit patterns, and team activity, allowing project managers to assess project health, allocate resources effectively, and address potential risks early on.

Complexity of Git Repositories: Modern software projects often involve multiple contributors working on different features and branches simultaneously. Navigating through the history of a Git repository to understand who contributed what and when can be overwhelming, especially for larger projects. Gitulyse simplifies this process by providing intuitive visualisations and detailed metrics that enable developers and project managers to gain a clear understanding of the repository's evolution.

Optimising Development Workflows: Inefficient development workflows can lead to wasted time and effort, ultimately delaying project delivery. By analysing commit patterns, code contributions, and collaboration dynamics, Gitulyse helps identify bottlenecks and inefficiencies in the development process. Armed with these insights, teams can streamline their workflows, allocate resources more effectively, and deliver software faster without compromising quality.

Continuous Improvement: Continuous improvement is essential for staying competitive in today's rapidly evolving technology landscape. Gitulyse enables organisations to track key metrics related to code quality, team productivity, and project health over time. By monitoring these metrics and identifying trends, teams can iteratively improve their development processes, refine their coding practices, and deliver higher-quality software with each iteration.

Research

The development of Gitulyse required extensive research across various disciplines, including software engineering, data analysis, and user experience design. Key areas of research included:

Version Control Systems (VCS): In-depth knowledge of Git's data model, branching strategies, and merge patterns was essential for building Gitulyse's analysis engine. Research in this area involved studying Git internals, understanding how Git stores and tracks changes, identifying the most efficient ways to extract and process data from Git repositories, and an in-depth study of the GitHub Flask API.

Data Manipulation and Analysis: Gitulyse relies on advanced algorithms and techniques from the field of data manipulation and analysis to extract meaningful insights from Git repositories. Research in data manipulation involved developing algorithms for detecting code contributions and analysing collaboration dynamics. These algorithms were designed to scale efficiently to handle repositories of any size.

Visualisation Techniques: Visualisations play a crucial role in conveying complex information. Research in visualisation techniques informed the design of Gitulyse's user interface, including the choice of charts, graphs, and interactive gauges. The goal was to create visualisations that are both informative and easy to understand, allowing users to explore and interpret repository/user data effortlessly.

Software Metrics and Quality Assurance: Understanding software metrics and quality assurance principles was essential for designing Gitulyse's reporting capabilities. Research in this area involved identifying key metrics for assessing team productivity, and project health and developing algorithms for calculating these metrics accurately. The goal was to provide users with actionable insights that they could use to improve their development processes.

User Experience Design: Finally, research in user experience design played a crucial role in shaping Gitulyse's interface and user interaction patterns. The research involved conducting user interviews and usability testing to ensure that Gitulyse meets the needs of its target audience effectively. The goal was to create a user-friendly interface that makes it easy for users to navigate, explore, and interact with repository/user data.

In summary, Gitulyse is the result of extensive research and development efforts aimed at addressing the challenges faced by software development persons and teams in managing Git repositories effectively. By combining insights from various research disciplines, Gitulyse provides a comprehensive solution for repository analysis that enables organisations to optimise their development workflows,

enhance collaboration and communication, and drive continuous improvement in their software projects.

Use Cases

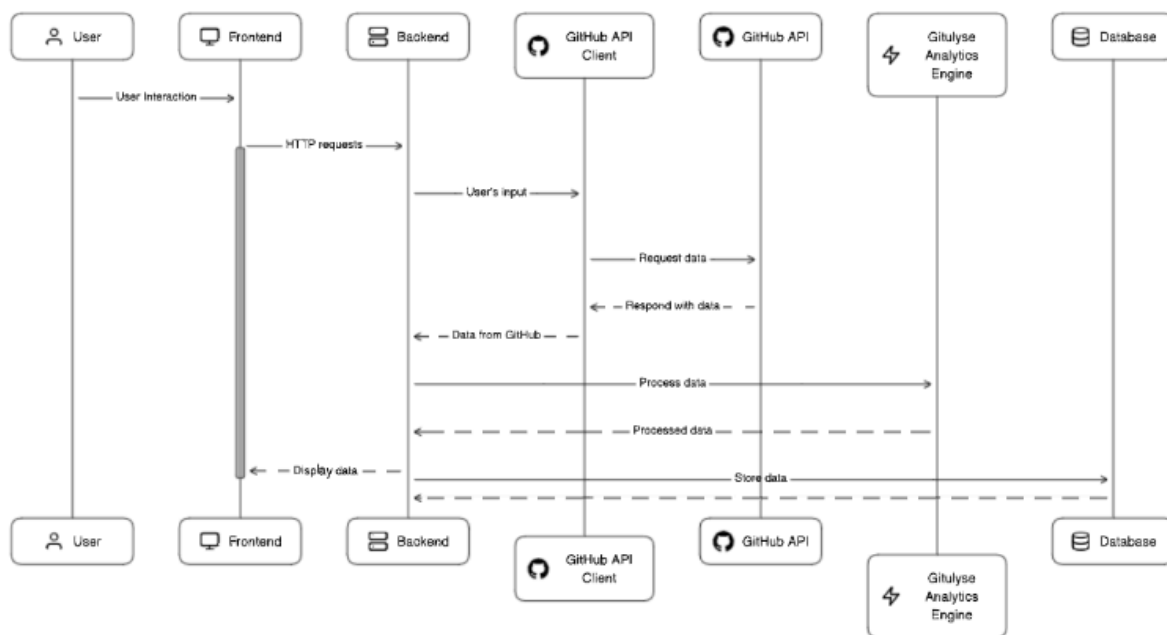
With our motivation for the project and our extensive research, we ended up with these 3 main use cases for our project:

- Gitulyse acts as an invaluable tool for those with limited technical know-how to analyse the progress of a GitHub repository. Through user-friendly visualisations and clear metrics, it allows users to understand the project's evolution, track commit patterns, and monitor overall activity, all without requiring extensive technical expertise.
- As a comprehensive Git CV, Gitulyse provides prospective employers with valuable insights into a candidate's projects and working practices. Generating detailed summaries, visual representations, and metrics from GitHub repositories, enables employers to evaluate a candidate's contributions, coding practices, and collaboration abilities, offering a holistic perspective on their skills and experiences.
- For individuals in managerial positions, Gitulyse serves as a valuable tool for comparing the performance and responsiveness of employees involved in GitHub repositories. By analysing metrics such as code contributions, and response times, managers can assess the productivity and effectiveness of team members, identify high-performing/low-performing individuals, and address any challenges or obstacles that may arise during the development process.

Design

High-Level Design

Sequence Diagram



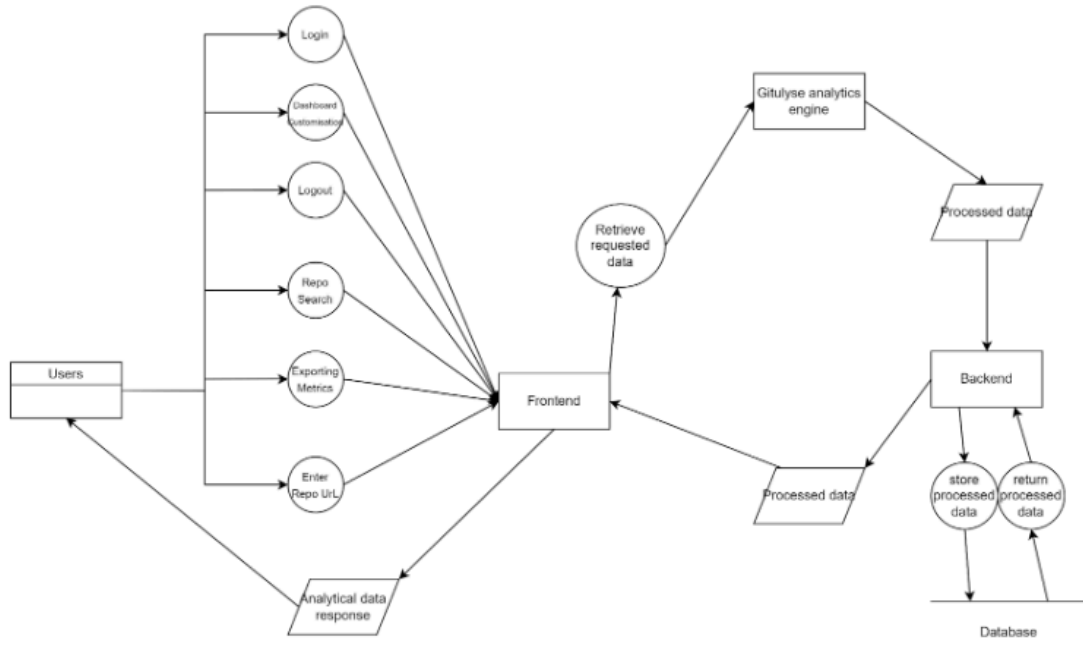
The sequence diagram illustrates the interactions between different components or objects in a system over a specific period. The above sequence diagram demonstrates the process of a user querying and receiving analytics data for a specific GitHub repository. Gitulyse has maintained this sequence throughout its development.

Explanation of the above diagram:

- A user interacts with the Gitulyse web application frontend through their browser
- Upon successful authentication, the backend/API server sends a request to the GitHub API client for repository data.
- The GitHub API client interacts with the GitHub API to retrieve the requested repository data.
- The GitHub API responds with the corresponding repo data, which is sent to the GitHub API client
- The GitHub API client passes the repository data to the backend/API server.
- The backend/API server processes and parses the data using the analytics engine.
- The backend/API server sends the analytical data response from the backend to the frontend.

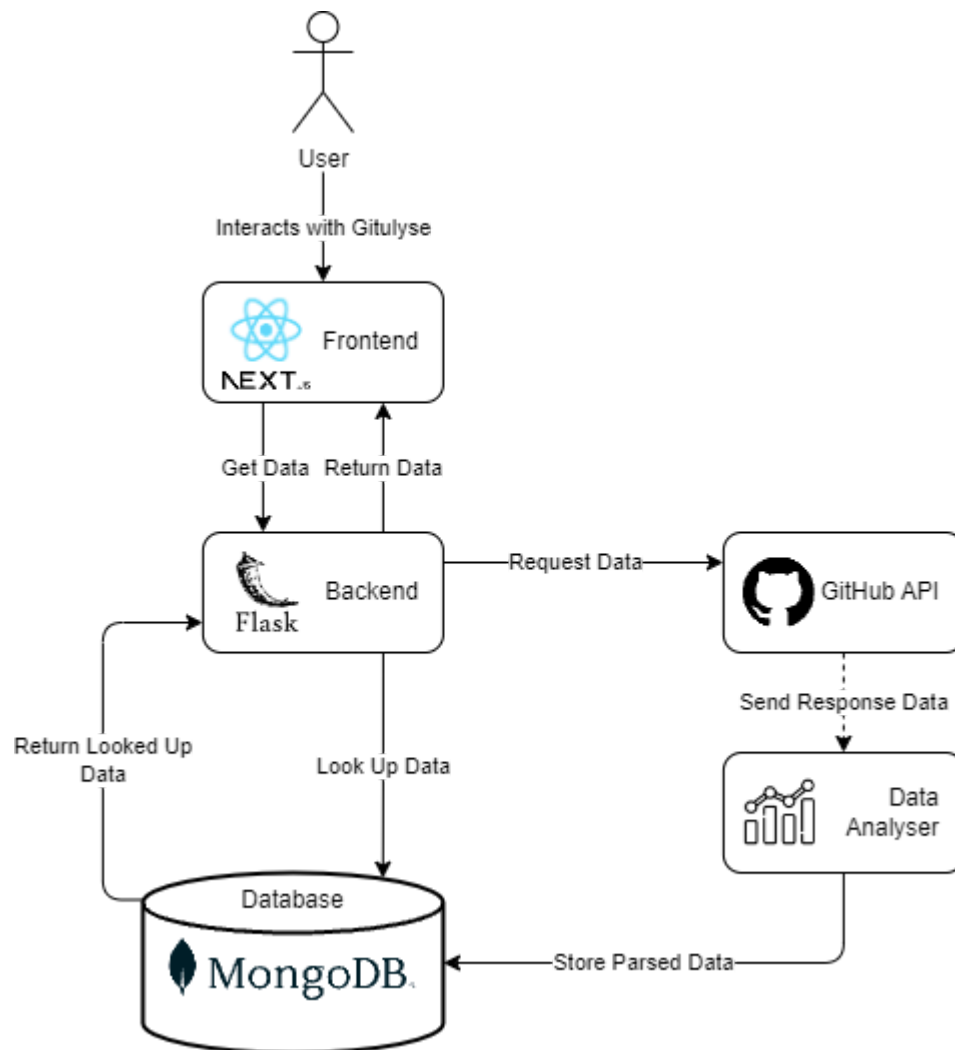
- The front end displays the analytics data to the user on the web application front end.

Data Flow Diagram



The data flow diagram maps out the flow and routes of information in the Gitulyse system. The above data flow diagram gives insight into the processes, products, Entities, and data storage in the Gitulyse system.

System Architecture



Gitulyse implements a 3 layer architecture as is the case with most modern architecture:

React Frontend:

Gitulyse features an intuitive and user-friendly interface crafted with Next.js, a potent React framework. Leveraging Next.js allows for seamless client-side rendering, enabling faster page loads and improved performance. Additionally, React's modular structure facilitates the creation of reusable components, ensuring consistency across the application and enhancing maintainability.

Flask Backend:

The backend infrastructure of Gitulyse is built upon Flask, a lightweight and flexible Python web framework. Flask excels at handling data processing tasks and facilitates seamless communication with the GitHub API, enabling Gitulyse to retrieve and manipulate data efficiently. By harnessing Flask's simplicity and

scalability, Gitulyses ensures a smooth and responsive user experience, even when dealing with complex operations.

Mongo Database:

Gitulyses relies on MongoDB as its primary database solution, offering unparalleled scalability and flexibility for data storage and management. MongoDB's document-oriented architecture aligns perfectly with Gitulyses' data model, allowing for efficient handling of diverse data types and structures. Moreover, MongoDB's distributed nature ensures high availability and fault tolerance, crucial for maintaining the integrity and reliability of Gitulyses' data infrastructure. By leveraging MongoDB, Gitulyses ensures optimal performance and scalability, catering to the evolving needs of its users.

Changes to the Original Architecture

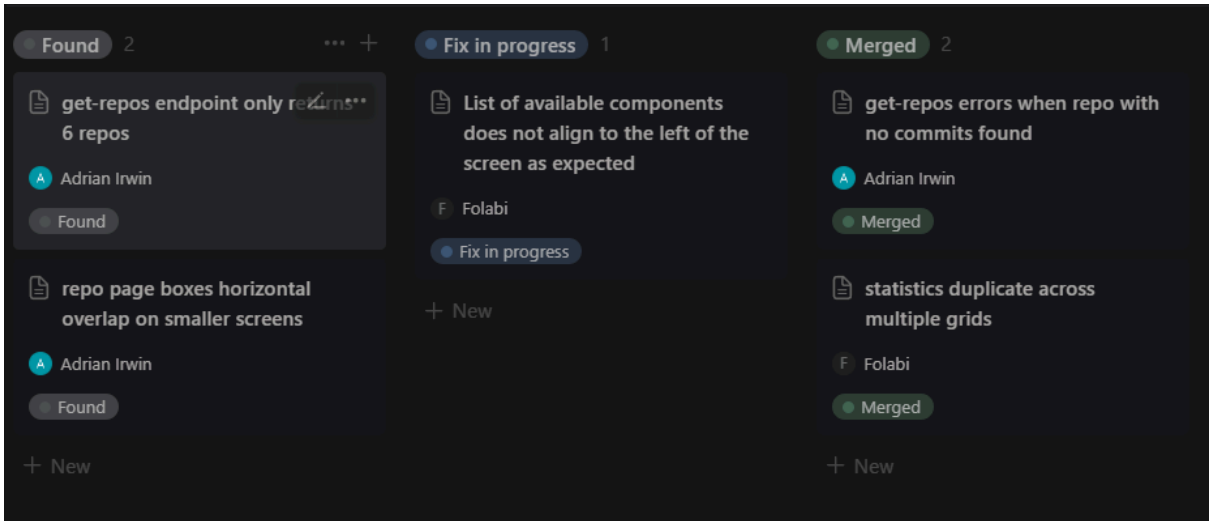
We have implemented a change in the database instead of the system architecture defined in the functional specification. This went from being a combination of Redis, and Prometheus (time series). We made this change to MongoDB as it has inbuilt facilities for storing time-series data, which would eliminate the original need for two databases.

In summary, the design choices behind Gitulyses' technology stack, including Next.js for the frontend, Flask for the backend, and MongoDB for the database, are carefully curated to deliver a seamless, responsive, and scalable platform for analysing and visualising GitHub repositories

Implementation

Bug Tracking

To ensure good workflow, we tracked our bugs using Notion. An example of our daily bug logs can be found below.



The following is a list of our recent bugs, the days they were found, and the days they were resolved.

Aa Name	Assign	Date Fixed	Date Found	Status
The floating action button used for creating	Adrian Irwin	16/04/2024	13/03/2024	Merged
The floating action button used for creating	Folabi	18/04/2024	11/03/2024	Merged
Graphs displayed in the info page extend be	Folabi	18/04/2024	13/03/2024	Merged
On the user overview page, the sidebar over	Folabi	18/04/2024	14/02/2024	Merged
repo page boxes horizontal overlap on s	Adrian Irwin	04/04/2024	02/04/2024	Merged
List of available components does not al	Folabi	04/04/2024	02/04/2024	Merged
statistics duplicate across multiple grids	Folabi	02/04/2024	30/03/2024	Merged
get-repos endpoint only returns 6 repos	Adrian Irwin	30/03/2024	30/03/2024	Merged
get-repos errors when repo with no com	Adrian Irwin	28/03/2024	29/03/2024	Merged

Continuous Integration

Continuous Integration is a vital practice in modern software development, ensuring that all changes made to the codebase are regularly tested, integrated, and deployed. Gitulyse provides a streamlined CI setup tailored to its specific requirements, aiming to enhance productivity and maintain code quality.

Defaults

By default all jobs use the node image by default as the majority of jobs ran are frontend based which all run under node. This does get overwritten in the backend jobs.

Stages

Gitulyse CI is structured into two distinct stages, each serving a specific purpose in the development lifecycle:

- **Lint:** Perform static code analysis to identify and report potential issues in the codebase.
- **Test:** Execute comprehensive tests to validate the functionality and integrity of the application.

Jobs

`.standard-rules:`

The purpose of this job is to define rules that define when jobs can be ran. In this scenario the jobs will only run in a merge request or if the branch is the default branch (main). This job gets reused by all the other jobs, through the extends keyword, so they all run with the same rules.

Frontend Lint

Stage: Lint

This job focuses on linting the frontend codebase to enforce code quality standards and identify potential issues early in the development process.

- **Before Script:** Sets up the environment by installing dependencies required for linting and navigating to the frontend directory.
- **Script:** Executes ESLint to analyse the code and report any violations.
- **Coverage:** Defines the regex pattern to extract and display the total test coverage percentage.
- **Artefacts:** Collects and stores the coverage report for further analysis.

Backend Test

Stage: Test

This job is responsible for running comprehensive tests on the backend codebase to verify its functionality and assess test coverage.

- Image: Specifies the Python Docker image to allow for the python tests to run.
- Before Script: Prepares the environment by installing dependencies.
- Script: Executes backend tests using pytest and generates a coverage report.
- Coverage: Defines the regex pattern to extract and display the total test coverage percentage.
- Artefacts: Collects and stores the coverage report for further analysis.

Conclusion

Gitulyse's Continuous Integration setup streamlines the development process by automating key tasks such as linting, testing, building, and deploying. By adhering to best practices and leveraging automation, Gitulyse ensures code quality, reliability, and efficiency throughout the software development lifecycle.

This technical guide provides a comprehensive overview of Gitulyse's CI configuration, empowering developers to leverage its capabilities effectively and drive continuous improvement in software delivery processes.

```
1  stages:
2    - lint
3    - test
4
5  default:
6    image: node
7
8  .standard-rules:
9    rules:
10     - if: $CI_PIPELINE_SOURCE == 'merge_request_event'
11     - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
12
13  frontend-lint:
14    stage: lint
15    extends: .standard-rules
16    before_script:
17     - echo "Installing dependencies..."
18     - cd src/gitulyse
19     - npm i
20     - echo "Dependencies installed."
21    script:
22     - echo "Running eslint..."
23     - npm run lint
24     - echo "EsLint complete."
25
```

```

26 backend-test:
27     stage: test
28     image: python:3.12.2
29     extends: .standard-rules
30     before_script:
31         - echo "Installing dependencies..."
32         - cd src/backend
33         - pip install -q -r requirements.txt
34         - pip install -q pytest-cov
35         - pip install -q -e .
36         - echo "Dependencies installed."
37     script:
38         - echo "Running backend tests..."
39         - pytest --cov --cov-report term --cov-report xml:coverage.xml
40         - echo "Backend tests complete."
41     coverage: '/TOTAL.*\s+(\d+%)$/'
42     artifacts:
43         when: always
44         reports:
45             coverage_report:
46                 coverage_format: cobertura
47                 path: src/backend/coverage.xml
48
49 frontend-test:
50     stage: test
51     extends: .standard-rules
52     before_script:
53         - echo "Installing dependencies..."
54         - cd src/gitulyse
55         - npm i
56         - echo "Dependencies installed."
57     script:
58         - npm run test -- --ci --coverage
59     coverage: /All files\^[^]*\^[^]*\s+([\d\.]+)/
60     artifacts:
61         when: always
62         reports:
63             coverage_report:
64                 coverage_format: cobertura
65                 path: src/gitulyse/coverage/cobertura-coverage.xml

```


Features

Login

The Login feature is essential for user authentication and access control within Gitulyse. It allows users to securely log in to their accounts, providing a personalised experience tailored to their preferences and permissions. The implementation of the Login feature involves:

- The login functionality in Gitulyse is implemented using client-side rendering, allowing for dynamic updates and interactivity without requiring full page reloads.
- Gitulyse utilises NextAuth's `useSession` hook to manage user sessions efficiently.
- Upon successful login, Gitulyse retrieves the user's access token from the session and stores it using React's `useState` hook for future API calls and authentication.

```
import { getSession, useSession } from "next-auth/react";
import { useEffect, useState } from "react";
import Marquee from "react-fast-marquee";

GoCodeo-Generate tests for the below function
const Home = () => {
  const { status } = useSession();
  const [user, setUser] = useState();

  const [userAccessToken, setUserAccessToken] = useState("");

  useEffect(() => {
    async function getInfo() {
      const info = await getSession();
      if (info) {
        setUser(info.login);
        setUserAccessToken(info.accessToken);
      }
    }

    getInfo().catch((err) => {
      console.error(err);
    });
  }, []);
```

- The `getSession` function is called asynchronously to retrieve user session information, including the user's login details.
- The UI is conditionally rendered based on the user's authentication status. If the user is authenticated, the Calendar and Repos components are displayed, providing access to Git repository analytics. Otherwise, a marquee with welcoming messages and a prompt to sign in with GitHub is displayed.

```

return (
  <section className="w-full flex-center flex-col">
    {status === "authenticated" ? (
      <>
        <Calendar userAccessToken={userAccessToken} user={
          user} />
        <Repos />
      </>
    ) : (

```

Logout

The Logout feature complements the Login functionality by enabling users to securely log out of their accounts and terminate their sessions. Key aspects of the Logout feature include:

- Similar to the login functionality, the signout function in Gitulyse is implemented using client-side rendering, allowing for seamless interaction and updates without full-page reloads.
- Gitulyse utilises NextAuth's useSession hook to manage user sessions and authentication state effectively.

```

"use client";
import { signIn, signOut, useSession } from "next-auth/react";

```

- The signOut function provided by NextAuth is invoked when the user clicks the "Sign Out" button, triggering the termination of the user's session and clearing authentication-related data.
- The UI is conditionally rendered based on the user's authentication status. If the user is authenticated, options for signing out, accessing user-specific information, and interacting with the application are displayed. Otherwise, a "Sign In" button is displayed to prompt users to authenticate.

Repo Information Page

The Information Page is a central hub for accessing essential information and resources within Gitulyse. It provides users with valuable insights, documentation, and support materials related to Git repositories and project management. Key elements of the Information Page include:

- **Client-Side Rendering:** The RepoPage component in Gitulyse is rendered on the client side, allowing for dynamic updates and interactions without full page reloads.
- **Session Management:** Gitulyse utilises NextAuth's getSession hook to manage user sessions and authentication tokens effectively.
- **Drag and Drop Functionality:** The component integrates drag-and-drop functionality using the react-dnd library, enabling users to organise and customise the displayed statistics interactively.

```
<DraggableNavItem name="Pull Requests" />
<DraggableNavItem name="Code Contributions" />
<DraggableNavItem name="Issue Tracking" />
<DraggableNavItem name="Percentage Pull Requests" />
<DraggableNavItem name="Percentage Issues" />
```

- **API Integration:** The component fetches repository statistics from the backend server using the fetch API, passing the user's access token and repository details as parameters.

```
useEffect(() => {
  if (!userAccessToken) return;

  fetch(`${BACKEND_URL}/get-repo-stats?token=${userAccessToken}&
owner=${owner}&repo=${repo}`)
    .then((res) => res.json())
    .then((data) => {
      setRepoInfo(data);
    });
}, [BACKEND_URL, owner, repo, userAccessToken]);
```

- **Statistical Visualisation:** Gitulyse presents a variety of repository statistics, including stars, forks, watchers, pull requests, languages, contributors, and their contributions, providing users with comprehensive insights into repository activity and engagement.

```

<Stack align="flex-start" justify="flex-start">
  <Title order={1}>Overall Stats</Title>
  <Group>
    <IconStar stroke={2} />
    <Text size="lg">Stars: {repoInfo?.stars}</Text>
  </Group>
  <Group>
    <IconGitFork stroke={2} />
    <Text size="lg">Forks: {repoInfo?.forks}</Text>
  </Group>
  <Group>
    <IconEye stroke={2} />
    <Text size="lg">Watchers: {repoInfo?.watchers}</Text>
  </Group>
  <Group>
    <IconGitPullRequest stroke={2} />
    <Text size="lg">
      Closed Pull Requests:{" "}
      {repoInfo?.pull_requests - repoInfo?.open_pull_requests}
    </Text>
  </Group>
  <Group>
    <IconGitPullRequestDraft stroke={2} />
    <Text size="lg">
      Open Pull Requests: {repoInfo?.open_pull_requests}
    </Text>
  </Group>
  <Group justify="flex-start" align="flex-start">
    <IconLanguage stroke={2} />
    <Stack>
      {repoInfo?.languages &&
        Object.entries(repoInfo?.languages)
          .sort(([, a], [, b]) => b - a)
          .map(([language, bytes]) => (
            <Group key={language}>
              <Text size="lg">
                {language}: {bytes}
              </Text>
            </Group>
          ))}
    </Stack>
  </Group>
</Stack>

```

- **Conditional Rendering:** The UI is conditionally rendered based on the availability of repository statistics. While waiting for the data to be fetched, a loader is displayed to indicate that the information is being loaded
- **Modular Components:** The RepoPage component comprises modular components responsible for displaying specific repository statistics, such as code contributions, pull requests, issue tracking, and percentage metrics.

```
const renderItem = (item) => {
  const { name } = item;
  switch (name) {
    case "Code Contributions":
      return (
        <CodeContributions
          owner={owner}
          repo={repo}
          userAccessToken={userAccessToken}
        />
      );
    case "Pull Requests":
      return <PullRequests owner={owner} repo={repo} userAccessToken={userAccessToken} />;
    case "Issue Tracking":
      return (
        <IssueTracking owner={owner} repo={repo} userAccessToken={userAccessToken} />
      );
    case "Percentage Pull Requests":
      return (
        <PercentagePullrequests
          start_date={startDate}
          end_date={endDate}
          owner={owner}
          repo={repo}
          userAccessToken={userAccessToken}
        />
      );
    case "Percentage Issues":
      return (
        <PercentageIssues
          start_date={startDate}
          end_date={endDate}
          owner={owner}
          repo={repo}
          userAccessToken={userAccessToken}
        />
      );
    default:
      return null;
  }
};
```

Modular Components

Gitulyse employs a modular architecture to enhance code organisation, maintainability, and reusability. Each modular component serves a specific function within the application, contributing to its overall functionality and user experience. This comprehensive documentation provides insights into the purpose, usage, and implementation details of the modular components in Gitulyse.

CodeContributions Component

The CodeContributions component displays statistics related to code contributions within a Git repository. It provides insights into the distribution of code contributions among contributors, highlighting their impact on the repository's codebase.

Backend Implementation:

- **parse_contributions** Function: This function parses contribution data obtained from the GitHub API response. It aggregates contributions by month and author, calculating additions, deletions, and commits.

```
def parse_contributions(contributor: StatsContributor, contributions, single_user=False):
    author = contributor.author
    weeks: list[StatsContributor.Week] = contributor.weeks
    for week in weeks:
        month = week.w.date().strftime("%Y-%m")
        if not single_user:
            if author.login not in contributions[month]:
                contributions[month][author.login] = {
                    "additions": week.a,
                    "deletions": week.d,
                    "commits": week.c,
                }
            else:
                contributions[month][author.login]["additions"] += week.a
                contributions[month][author.login]["deletions"] += week.d
                contributions[month][author.login]["commits"] += week.c
        else:
            contributions[month]["additions"] += week.a
            contributions[month]["deletions"] += week.d
            contributions[month]["commits"] += week.c

    return contributions
```

- **get_contributions_from_repo** Function: This function retrieves contribution data from a specified GitHub repository using the GitHub API. It fetches statistics contributors and aggregates their contributions using the `parse_contributions` function.

```
def get_contributions_from_repo(token, owner, repo_name):
    auth = Auth.Token(token)
    gh = Github(auth=auth)

    repo_full_name = f"{owner}/{repo_name}"

    repo = gh.get_repo(repo_full_name)

    contributors = repo.get_stats_contributors()
    contributions = {}
    for week in contributors[0].weeks:
        contributions[week.w.date().strftime("%Y-%m")] = {}

    for contributor in contributors:
        contributions = parse_contributions(contributor, contributions)

    return contributions
```

- /code-contribution-stats Route: This route is exposed for retrieving code contribution statistics. It expects 'token', 'owner', and 'repo' parameters in the request query string. It validates the parameters and handles exceptions, such as invalid tokens or missing owner/repo information.

```
@bp.route("/code-contribution-stats", methods=["GET"])
def code_contribution_stats():
    token = request.args.get("token")
    owner = request.args.get("owner")
    repo_name = request.args.get("repo")

    if owner is None or owner == "":
        return jsonify({"message": "No owner provided"}), 400
    else:
        owner = owner.lower()

    if repo_name is None or repo_name == "":
        return jsonify({"message": "No repo provided"}), 400
    else:
        repo_name = repo_name.lower()

    try:
        contributions = get_contributions_from_repo(token, owner, repo_name)
    except BadCredentialsException:
        return jsonify({"message": "Invalid token"}), 401
```

Frontend Implementation:

The CodeContributions component utilises the Backend to fetch and display statistics such as commits, lines of code added, lines of code deleted, and the overall code contribution graph. It leverages React components and a data visualisation library(recharts), to present these statistics in a visually appealing and informative manner.

```
fetch(`${BACKEND_URL}/code-contribution-stats?${queryParams}`)
  .then((res) => res.json())
  .then((data) => {
    // Extract all authors from the data
    const authors = Object.keys(data.monthly).reduce((acc,
    month) => {
      const monthAuthors = Object.keys(data.monthly[month]);
      return [...acc, ...monthAuthors];
    }, []);

    // Remove duplicates and set all authors
    setAllAuthors([...new Set(authors)]);

    // Set monthly data
    setMonthlyData(
      Object.entries(data.monthly).map(([month, authors]) =>
        ({
          month,
          ...authors,
        })),
    );
  });
}, [userAccessToken, BACKEND_URL, owner, repo]);
```



```

<p className="mb-4 text-2xl">Code Contributions per Month</p>
<LineChart width={600} height={300} data={monthlyData}>
  <CartesianGrid strokeDasharray="3 4" />
  <XAxis dataKey="month" />
  <YAxis />
  <Tooltip content={CustomChartTooltip} />
  {allAuthors.map((author, index) => (
    <Line
      key={author}
      type="monotone"
      dataKey={author}
      name={author}
      stroke={getColour(index)}
      activeDot={{ r: 8 }}
      isAnimationActive={false}
      hide={!selectedAuthors.includes(author)}
    />
  ))}
</LineChart>
<div className="mt-4">
  <button
    className="mr-2 mb-2 p-2 rounded bg-blue-500
    text-white"
    onClick={() => {
      if (selectedAuthors.length === allAuthors.length) {
        setSelectedAuthors([]);
      } else {
        setSelectedAuthors(allAuthors);
      }
    }}
  >
    Toggle All
  </button>
  {allAuthors.map((author, index) => (
    <button
      key={author}
      onClick={() => handleAuthorSelection(author)}
      className={`mr-2 mb-2 p-2 rounded ${
        selectedAuthors.includes(author) ? "" :
        "bg-gray-200 text-gray-500"
      }`}
      style={{
        backgroundColor: selectedAuthors.includes
          (author)
          ? getColour(index)
          : "",
      }}
    >
      {author}
    </button>
  ))}

```

IssueTracking Component

The IssueTracking component offers insights into issue management within a Git repository. It displays information on the trends of issue management through your repository lifecycle.

Backend Implementation:

- `calculate_percentage_issues_resolved` Function: Iterates through the list of issues and checks if each issue was closed within the specified date range. Increments the count of resolved issues accordingly.

```
def calculate_percentage_issues_resolved(issues, start_date, end_date):
    total_issues_resolved = 0
    for issue in issues:
        if issue.closed_at is not None:
            if start_date.timestamp() <= issue.closed_at.timestamp() <= end_date.timestamp():
                total_issues_resolved += 1
    return total_issues_resolved
```

- `/get-issues` Route: Fetches issues from the GitHub repository using the provided token, owner, and repo information. Calculates and includes additional information such as issue state, author, creation/update dates, and time to close (if closed).

```
@bp.route("/get-issues", methods=["GET"])
def get_issues():
    token = request.args.get("token")
    owner = request.args.get("owner")
    repo_name = request.args.get("repo")

    if owner is None or owner == "":
        return jsonify({"message": "No owner provided"}), 400
    else:
        owner = owner.lower()

    if repo_name is None or repo_name == "":
        return jsonify({"message": "No repo provided"}), 400
    else:
        repo_name = repo_name.lower()

    repo = f"{owner}/{repo_name}"

    auth = Auth.Token(token)
    gh = Github(auth=auth)

    try:
        repo = gh.get_repo(repo)
    except BadCredentialsException:
        return jsonify({"message": "Invalid token"}), 401
    issues = repo.get_issues(state="all", direction="asc")
```

Frontend Implementation:

The IssueTracking component interacts with the Backend to fetch and display information about repository issues. It utilises React components and a data visualisation library(recharts) to render a graph with the raised issues over the timeline of the project.

```
fetch(`${BACKEND_URL}/get-issues?token=${userAccessToken}&owner=${owner}&repo=${repo}`)
  .then((res) => res.json())
  .then((data) => {
    const { issues, average_time_to_resolve } = data;
    setData(issues);
    setAverageTimeToResolve(average_time_to_resolve);
  });
}, [userAccessToken, BACKEND_URL, owner, repo]);

const formatTime = (totalSeconds) => {
  const days = Math.floor(totalSeconds / 86400);
  const hours = Math.floor((totalSeconds % 86400) / 3600);
  const minutes = Math.floor((totalSeconds % 3600) / 60);
  const seconds = Math.floor(totalSeconds % 60);

  let formattedTime = '';
  if (days > 0) {
    formattedTime += `${days}d `;
  }
  if (hours > 0) {
    formattedTime += `${hours}h `;
  }
  if (minutes > 0) {
    formattedTime += `${minutes}m `;
  }
  formattedTime += `${seconds}s`;

  return formattedTime;
};
```

```
<div className="mt-3 flex flex-col items-center">
  <p className="mb-4 text-2xl">Issue Tracking</p>
  <div>
    <p>Average Time to Resolve Issues: {formatTime(averageTimeToResolve)}</p>
    <AreaChart width={600} height={300} data={data}>
      <CartesianGrid strokeDasharray="3 4" />
      <XAxis dataKey="issue_number" />
      <YAxis tickFormatter={(value) => largestFormatTime(value)} />
      <Tooltip formatter={(value) => formatTime(value)} />
      <Legend />
      <Area type="monotone" dataKey="time_to_close.total_seconds" stackId="1" stroke="#8884d8" fill="#8884d8" name="Time to Resolve" />
    </AreaChart>
  </div>
</div>
```

PullRequests Component

The PullRequests component provides visibility into the pull request activity within a Git repository. It showcases details about the time taken to merge pull requests.

Backend Implementation:

- `parse_pull_request` Function: Extracts relevant information such as title, author, creation/update dates, PR number, state (merged/closed), and time taken to merge (if merged). Calculates the time to merge if the PR is merged.

```
def parse_pull_request(pull_request):
    pull_request_info = {
        "title": pull_request.title,
        "author": pull_request.user.login,
        "created_at": pull_request.created_at,
        "updated_at": pull_request.updated_at,
        "pr_number": pull_request.number,
    }

    if pull_request.merged_at is not None:
        pull_request_info["state"] = "merged"
        pull_request_info["merged_at"] = pull_request.merged_at

        time_to_merge = (
            pull_request.merged_at - pull_request.created_at
        ).total_seconds()

        days, remainder = divmod(time_to_merge, 86400)
        hours, remainder = divmod(remainder, 3600)
        minutes, seconds = divmod(remainder, 60)

        time_to_merge_obj = {
            "days": int(days),
            "hours": int(hours),
            "minutes": int(minutes),
            "seconds": int(seconds),
            "total_seconds": int(time_to_merge),
        }

        pull_request_info["time_to_merge"] = time_to_merge_obj
    elif pull_request.closed_at is not None:
        pull_request_info["state"] = pull_request.state
        pull_request_info["closed_at"] = pull_request.closed_at
    else:
        pull_request_info["state"] = pull_request.state

    return pull_request_info
```

- /get-pull-requests Route: Fetches pull requests from the GitHub repository using the provided token, owner, and repo information. Utilises multithreading to concurrently parse pull request information using the parse_pull_request function. Constructs a list of pull request information dictionaries and returns them in the JSON response.

```
repo = f"{owner}/{repo_name}"

auth = Auth.Token(token)
gh = Github(auth=auth)

try:
    repo = gh.get_repo(repo)
except BadCredentialsException:
    return jsonify({"message": "Invalid token"}), 401
pull_requests = repo.get_pulls(state="all", direction="asc")
pull_request_list = []
with ThreadPoolExecutor(max_workers=20) as executor:
    for pull_request_info in executor.map(parse_pull_request, pull_requests):
        pull_request_list.append(pull_request_info)

return jsonify({"pull_requests": pull_request_list})
```

Frontend Implementation:

The PullRequests component interacts with the Backend to retrieve and present data related to repository pull requests. It parses the information down, and formats the time to readable time then, utilises React components and a data visualisation library(recharts) to render pull request details into an intuitive graph.

```
fetch(
    `${BACKEND_URL}/get-pull-requests?token=${userAccessToken}&owner=${owner}&repo=${repo}`,
)
    .then((res) => res.json())
    .then((data) => {
        setPullRequests(data.pull_requests);
    });
}, [userAccessToken, BACKEND_URL, owner, repo]);
```

```
const formatTime = (totalSeconds) => {
  const days = Math.floor(totalSeconds / 86400);
  const hours = Math.floor((totalSeconds % 86400) / 3600);
  const minutes = Math.floor((totalSeconds % 3600) / 60);
  const seconds = Math.floor(totalSeconds % 60);

  let formattedTime = '';
  if (days > 0) {
    formattedTime += `${days}d `;
  }
  if (hours > 0) {
    formattedTime += `${hours}h `;
  }
  if (minutes > 0) {
    formattedTime += `${minutes}m `;
  }
  formattedTime += `${seconds}s`;

  return formattedTime;
};
```

```
<div className="mt-3 flex flex-col items-center">
  <p className="mb-4 text-2xl">Time to Merge Pull requests</p>

  {pullRequests.length > 0 && (
    <BarChart width={600} height={300} data={timeToMergeData}>
      <CartesianGrid strokeDasharray="3 3" />
      <XAxis dataKey="name" tickFormatter={(value) => `PR ${value}`} />
      <YAxis tickFormatter={(value) => largestFormatTime(value)} />
      <Tooltip formatter={(value) => formatTime(value)} />
      <Legend />
      <Bar dataKey="time_to_merge" fill="#8884d8" name="Time to Merge" />
    </BarChart>
  )}
</div>
```

PercentagePullrequests Component

The PercentagePullrequests component calculates and displays the percentage of pull requests merged within a specified date range compared to the total number of pull requests opened within the time frame specified.

Backend Implementation:

- /get-percentage-pull-requests Route: Retrieves pull requests from the GitHub repository using the provided token, owner, and repo information. Filters pull

requests based on the specified date range. Calculates the total number of pull requests and the number of merged pull requests. Computes the percentage of merged pull requests and returns the result.

```
@bp.route("/get-percentage-pull-requests", methods=["GET"])
def get_percentage_pull_requests():
    token = request.args.get("token")
    owner = request.args.get("owner")
    repo_name = request.args.get("repo")
    start_date = request.args.get("start_date")
    end_date = request.args.get("end_date")

    if owner is None or owner == "":
        return jsonify({"message": "No owner provided"}), 400
    else:
        owner = owner.lower()

    if repo_name is None or repo_name == "":
        return jsonify({"message": "No repo provided"}), 400
    else:
        repo_name = repo_name.lower()

    if start_date is None or start_date == "":
        return jsonify({"message": "No start date provided"}), 400

    if end_date is None or end_date == "":
        return jsonify({"message": "No end date provided"}), 400
```

Frontend Implementation:

The PercentagePullrequests component communicates with the Backend to retrieve pull request data within the specified date range. It calculates the percentage of pull requests merged during that period and renders the result using React components (react-gauge-chart).

```
fetch(`${BACKEND_URL}/get-percentage-pull-requests?${queryParams}`)
  .then((response) => response.json())
  .then((data) => {
    console.log("Data received", data);
    setPercentage(data.percentage_merged);
  })
  .catch((err) => console.error(err));
}, [formSubmitted, userAccessToken, startDate, endDate, BACKEND_URL, owner, repo]);
```

```

<div className="mt-3 flex flex-col items-center">
  <p className="mb-4 text-2xl">Percent of PR's merged/created from</p>
  <form onSubmit={handleSubmit}>
    <label className="mr-3">
      Start Date:
      <input
        className="ml-1"
        type="date"
        value={startDate}
        onChange={(e) => setStartDate(e.target.value)}
      />
    </label>
    <label>
      End Date:
      <input
        className="ml-1"
        type="date"
        value={endDate}
        onChange={(e) => setEndDate(e.target.value)}
      />
    </label>
    <Button type="submit" className="ml-3">
      Submit
    </Button>
  </form>
  {percentage !== null && (
    <div style={{ width: "400px", margin: "20px auto" }}>
      <GaugeChart
        id="gauge-chart1"
        percent={percentage / 100}
        colors={["red", "yellow", "green"]}
      />
    </div>
  )}
</div>

```

PercentageIssues Component

The PercentageIssues component computes and visualises the percentage of resolved issues compared to the total number of issues opened in a Git repository within a specified time frame.

Backend Implementation:

/get-percentage-issues Route: Fetches issues from the GitHub repository using the provided token, owner, and repo information. Filters issues based on the specified date range. Calculates the total number of issues and the total number of resolved issues using the calculate_percentage_issues_resolved function. Computes the percentage of issues resolved and returns the result.

Frontend Implementation:

The PercentageIssues component interacts with the Backend to fetch issue data within the designated date range. It then calculates the percentage of resolved issues relative to the total number of issues opened and presents the result using the React component(react-gauge-chart).

```
fetch(`${BACKEND_URL}/get-percentage-issues?${queryParams}`)
  .then((response) => response.json())
  .then((data) => {
    console.log("Data received", data);
    setPercentage(data.percentage_issues_resolved);
  })
  .catch((err) => console.error(err));
}, [formSubmitted, userAccessToken, startDate, endDate, BACKEND_URL, owner, repo]);
```

```
<div className="mt-3 flex flex-col items-center">
  <p className="mb-4 text-2xl">Percentage of issues resolved/created from</p>
  <form onSubmit={handleSubmit}>
    <label className="mr-3">
      Start Date:
      <input
        className="ml-1"
        type="date"
        value={startDate}
        onChange={(e) => setStartDate(e.target.value)}
      />
    </label>
    <label>
      End Date:
      <input
        className="ml-1"
        type="date"
        value={endDate}
        onChange={(e) => setEndDate(e.target.value)}
      />
    </label>
    <Button type="submit" className="ml-3">
      Submit
    </Button>
  </form>
  {percentage !== null && (
    <div style={{ width: "400px", margin: "20px auto" }}>
      <GaugeChart
        id="gauge-chart1"
        percent={percentage / 100}
        colors={["red", "yellow", "green"]}
      />
    </div>
  )}
</div>
```

User Page

The User Page offers users a personalised dashboard and workspace within Gitulyse, empowering them to manage their projects, repositories, and collaboration activities efficiently. Implementation of the user page is as follows:

Frontend Implementation

This React component, `UserPage`, fetches and displays GitHub user information. It imports necessary modules and components, initialises state variables for managing data, and utilises effect hooks to fetch data and update the UI accordingly. It renders a loading overlay while data is being fetched, and then displays user information including base info, contribution charts, and a list of repositories.

```
<Container size="xl" pos="relative">
  <LoadingOverlay
    visible={isLoading}
    zIndex={1000}
    overlayProps={{ radius: "sm", blur: 12 }}
  />
  <Stack className="mt-7" gap="xs" align="stretch" justify="space-between">
    <BaseInfo userInfo={userInfo} />

    <ContributionChart
      data={rechartsData}
      largest={largestAverageContributions}
      userInfo={userInfo}
    />

    <Center mt="lg" mb="md">
      <Title order={3}>Repositories</Title>
    </Center>
    <RepoList repos={userInfo.repos} />
  </Stack>
</Container>
```

Backend Implementation

/get-user:

- Extracts token, user, and force parameters from the request query.
- Attempts to authenticate with the provided token and retrieve user information using the GitHub API.
- Handles cases for invalid tokens and unknown users, returning appropriate error messages.
- Check the local database for cached user information to avoid excessive API calls.
- Retrieves and constructs detailed user information, including:
- User profile details like login, name, email, location, bio, public repository count, created at the timestamp, avatar URL, and HTML URL.

- List of user's repositories with names and HTML URLs.
- Contribution statistics for each repository and overall contributions, including additions, deletions, and commits per month.
- Languages used across all user repositories with byte counts.
- Number of pull requests authored by the user.
- Cleans up contributions data to remove months prior to the user's account creation.
- Utilises multithreading for concurrent processing of repository data to improve performance.

```
@bp.route("/get-user", methods=["GET"])
def get_user():
    token = request.args.get("token")
    user = request.args.get("user")
    force = request.args.get("force")

    db_client = get_db()

    try:
        auth = Auth.Token(token)
        g = Github(auth=auth)

        if user:
            user = g.get_user(user.lower())
        else:
            user = g.get_user()
    except BadCredentialsException:
        return jsonify({"message": "Invalid token"}), 401
    except UnknownObjectException:
        return jsonify({"message": "User not found"}), 404

    try:
        latest_user_info = db_client["users"][user.login.lower()].find({}, {"_id": 0}).sort("timestamp", -1).limit(1)[0]
        if not force and latest_user_info["timestamp"] > time() - 600:
            return jsonify(latest_user_info), 200
    except IndexError:
        pass

    user_info = {
        "login": user.login,
        "name": user.name,
        "email": user.email,
        "location": user.location,
        "bio": user.bio,
        "public_repo_count": user.public_repos,
        "created_at": user.created_at,
        "avatar_url": user.avatar_url,
        "html_url": user.html_url,
        "repos": {},
        "languages": get_user_languages(user.get_repos()),
        "pull_request_count": len([pr for pr in g.search_issues(f"author:{user.login} is:pr")]),
    }

    all_user_repos = [repo for repo in user.get_repos(type="all")]
    user_info["repos"] = [{"name": repo.full_name, "html_url": repo.html_url} for repo in all_user_repos]

    contributions = get_user_contributions(all_user_repos, user)
```

clean_contributions: Removes contributions data for months before the user's account creation.

```
def clean_contributions(contributions, user_created_at):
    user_created_at = user_created_at - relativedelta(months=1)
    to_pop = []
    for month in contributions.keys():
        converted_month = datetime.strptime(month, "%Y-%m")
        if converted_month.timestamp() < user_created_at.timestamp():
            to_pop.append(month)

    for month in to_pop:
        contributions.pop(month)

    return contributions
```

process_repo: Processes contribution statistics for a single repository.

```
def process_repo(repo, user, contributions_per_repo, overall_contributions):
    contributions_per_repo[repo.full_name] = {}

    stats_contributors = repo.get_stats_contributors()
    if stats_contributors is None:
        return

    for week in stats_contributors[0].weeks:
        week_formatted = week.w.date().strftime("%Y-%m")
        contributions_per_repo[repo.full_name][week_formatted] = {
            "additions": 0,
            "deletions": 0,
            "commits": 0,
        }
        if week_formatted not in overall_contributions:
            overall_contributions[week_formatted] = {
                "additions": 0,
                "deletions": 0,
                "commits": 0,
            }

    user_stats_contributor = next(
        (contributor for contributor in stats_contributors if contributor.author.login == user.login), None)

    with ThreadPoolExecutor(max_workers=2) as parse_executor:
        parse_executor.submit(parse_contributions, user_stats_contributor, contributions_per_repo[repo.full_name],
                               True)
        parse_executor.submit(parse_contributions, user_stats_contributor, overall_contributions, True)
```

get_user_contributions: Retrieves contribution statistics for all user repositories.

```
def get_user_contributions(all_user_repos, user):
    contributions_per_repo = {}
    overall_contributions = {}

    with ThreadPoolExecutor(max_workers=5) as executor:
        _ = [executor.submit(process_repo, repo, user, contributions_per_repo, overall_contributions) for repo in
              all_user_repos]

    overall_contributions = clean_contributions(overall_contributions, user.created_at)
    for repo in contributions_per_repo.keys():
        contributions_per_repo[repo] = clean_contributions(contributions_per_repo[repo], user.created_at)

    return {
        "repo_contributions": contributions_per_repo,
        "overall_contributions": overall_contributions
    }
```

get_user_languages: Retrieves languages used across all user repositories.

```
def get_user_languages(all_user_repos):  
    languages = {}  
    with ThreadPoolExecutor(max_workers=5) as executor:  
        for repo in all_user_repos:  
            executor.submit(process_repo_languages, repo, languages)  
  
    return languages
```

Comparison Page

The Comparison Page, not included in the functional spec, enables users to compare different users within Gitulyse. Extra information is also provided on both users, such as their total pull requests opened and languages used across the lifetime of their GitHub accounts, facilitating the decision-making processes of a prospective employer or person in a management position.

Key functionalities of the Comparison Page include the:

Backend Implementation

Note* The Comparison page shares the same backend implementation as the user page except it is called twice to compare two separate users.

Frontend Implementation

The UserComparePage component is involved with the comparison of two GitHub users, offering insights into their contributions over time. Through state management using useState hooks, it maintains variables such as user access tokens, user information for both individuals and chart data representing their contributions. Two use-effect hooks handle the asynchronous fetching of user data from a backend server upon receiving access tokens, ensuring the component displays up-to-date information. Additional useEffect hooks update the chart data and loading states based on the fetched user information, enabling seamless rendering of the comparison interface.

```

useEffect(() => {
  if (!userAccessToken) return;

  fetch(`${BACKEND_URL}/get-user?token=${userAccessToken}&user=${user_one}`)
    .then((res) => res.json())
    .then((data) => {
      setUserOneInfo(data);
      disableLoadingUserOne();
    });

  fetch(`${BACKEND_URL}/get-user?token=${userAccessToken}&user=${user_two}`)
    .then((res) => res.json())
    .then((data) => {
      setUserTwoInfo(data);
      disableLoadingUserTwo();
    });
}, [
  BACKEND_URL,
  disableLoadingUserOne,
  disableLoadingUserTwo,
  userAccessToken,
  user_one,
  user_two,
]);

```

Rendering the comparison page involves rendering loading overlays to signify data fetching periods and displaying a descriptive title indicating the comparison between the two users. The main content consists of two instances of the `SingleUserCompare` component arranged side by side, each receiving user information and chart data as props. These instances facilitate the visual comparison of the users' contributions, offering insights into their activity trends over time.

```

<Box className="max-w-full mt-6" pos="relative">
  <LoadingOverlay
    visible={overallLoading}
    zIndex={1000}
    overlayProps={{ radius: "sm", blur: 12 }}
  />
  <Center className="pb-5">
    <Title order={1}>
      Comparison of &apos;
      {userOneInfo.name ? userOneInfo.name : userOneInfo.login}&apos; and &apos;
      {userTwoInfo.name ? userTwoInfo.name : userTwoInfo.login}&apos;
    </Title>
  </Center>

  <Box className="flex flex-row justify-center">
    <SingleUserCompare
      userInfo={userOneInfo}
      chartData={userOneChartData}
      position="left"
    />

    <Divider orientation="vertical" size="xl" />

    <SingleUserCompare
      userInfo={userTwoInfo}
      chartData={userTwoChartData}
      position="right"
    />
  </Box>
</Box>

```

Problems Resolved

Data Extraction and Processing Challenges:

Git repositories house vast amounts of data, ranging from commit history to issue practices and personal information. Extracting and processing this data efficiently posed a significant challenge. However, using concurrent techniques like 'Multithreading', and downscaling the amount of data we requested from the GitHub API we overcame these challenges with sluggish and inefficient data extraction and processing.

Visualisation Design and Usability:

Transforming raw Git data into meaningful visualisations required careful consideration of design principles and user experience. Designing intuitive interfaces to navigate through extensive commit histories, branch structures, and code contributions posed a significant usability challenge. Balancing simplicity with functionality, and providing customisable visualisation options to cater to diverse user preferences, demanded iterative design iterations and user feedback loops.

Additionally, choosing the right visualisation technique for the respective data, posed its challenges. However, through peer review techniques and extensive research, we were able to quickly retrieve, parse data, and choose appropriate visualisation techniques respectively.

Results

Testing Strategy

For Gitulyse, a comprehensive testing strategy incorporating both backend and frontend unit testing, ad-hoc testing, and user testing would ensure robustness and reliability across the application. Here's a breakdown of each component:

Backend Integration Testing with Pytest (API Endpoints):

- Utilised pytest, a popular testing framework for Python, to write integration tests for each API endpoint.
- Aim for full coverage by testing various scenarios including valid inputs, edge cases, and error conditions.
- Mock external dependencies and database interactions to isolate the tests and ensure they run reliably and efficiently.

Frontend Unit Testing with Jest:

- Leverage Jest, a widely-used testing framework for JavaScript, to write unit tests for frontend components.
- Test React and external components, to ensure they behave as expected.
- Mock API calls and external dependencies to isolate component tests and make them deterministic.
- Aim for wide coverage, testing different states and user interactions.

Ad-hoc Testing:

- Conduct ad-hoc testing to explore the application for usability issues, unexpected behaviour, and edge cases.
- This can be performed manually by testers.
- Focus on areas not covered by automated tests, such as UI layout, user experience, and non-functional requirements.

User Testing:

- Involve real users to perform user testing and gather feedback on the application's usability and functionality.
- Define test scenarios and user personas to guide the testing process.
- Collect feedback through a survey to identify pain points and areas for improvement.
- Iterate on the feedback received to refine the application and enhance user satisfaction.

By combining these testing approaches, Gitulyse ensures both functional correctness and user satisfaction, delivering a high-quality product that meets the needs of its users. Regular integration of feedback from ad-hoc testing and user testing into the development process will contribute to continuous improvement and refinement of the application.

Frontend Unit Testing with Jest

The following will outline the unit testing strategy of the Next.js front end using the Jest framework. In the Gitulyse context, Jest is a powerful tool for writing and executing unit tests. Here, we outline our unit testing strategy for the Next.js front end using the Jest framework.

Unit Test Coverage

Below is a coverage report for our frontend Unit tests:

All files

83.6% Statements 2876/3483 79.91% Branches 179/224 61.25% Functions 49/80 83.6% Lines 2876/3483

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements
app	92.18% 59/64
app/repo/[owner]/[repo]	56.23% 194/345
app/user/[user]	95.69% 89/93
app/user/[user]/[user_two]	96.29% 130/135
components	90.52% 621/686
components/repo	63.73% 290/455
components/user	98.18% 649/661
testing-utils	100% 44/44

Branches	Functions	Lines
80% 4/5	100% 2/2	92.18% 59/64
45.45% 10/22	44.44% 4/9	56.23% 194/345
85.71% 12/14	100% 3/3	95.69% 89/93
84% 21/25	100% 2/2	96.29% 130/135
87.5% 63/72	65.38% 17/26	90.52% 621/686
53.84% 14/26	22.72% 5/22	63.73% 290/455
90.56% 48/53	100% 11/11	98.18% 649/661
100% 7/7	100% 5/5	100% 44/44

- `HomePage.test.js`: Tests relating to rendering the homepage.
- `nav.test.js`: Tests relating to rendering components on the Navigation bar.
- `repoComponents.test.js`: Tests concerned with rendering the draggable components on the repository information page.

- RepoPage.test: Tests to verify the rendering of the repository information page itself.
- UserComparePage.test.js: Tests to verify the rendering of the User Comparison Page.
- userComponents.test.js: Test concerned with the rendering of the components on the User Comparison and User Information Pages.
- UserPage.test.js: Tests to verify the rendering of the User information page.

Test Environment Setup

To run the tests we set up the Jest testing environment using the following configurations:

jest.config.js:

```

1  const nextJest = require("next/jest");
2  /**
3   * For a detailed explanation regarding each configuration property, visit
4   * https://jestjs.io/docs/configuration
5   */
6
7  /** @type {import("jest").Config} */
8  const createJestConfig = nextJest({
9    dir: "./",
10  });
11
12  const config = {
13    clearMocks: true,
14    collectCoverage: true,
15    coverageDirectory: "coverage",
16    coverageProvider: "v8",
17    moduleDirectories: ["node_modules", "<rootDir>/"],
18    moduleNameMapper: {
19      "^@/(.*)/$": "<rootDir>/src/$1",
20    },
21    setupFilesAfterEnv: [<rootDir>/jest.setup.js"],
22    testEnvironment: "jsdom",
23    transformIgnorePatterns: ["/node_modules/(?!(d3)/)"],
24  };
25
26  module.exports = createJestConfig(config);

```

jest.setup.js:

```
import "@testing-library/jest-dom";
```

Test Cases

HomePage.test.js

- **renders correctly when the user is authenticated:** This test verifies that the Home component renders correctly when the user is authenticated. It mocks the authenticated session status using `useSession` and ensures that specific text elements are present in the rendered component.
- **renders correctly when the user is not authenticated:** Ensures that the Home component renders correctly when the user is not authenticated. It mocks the unauthenticated session status and checks for specific text elements related to signing in.

nav.test.js

- **renders without crashing:** Verifies that the Nav component renders without errors. It mocks the session status using `useSession` and ensures that the component renders the "Gitulyse" text.
- **renders sign in button when session is null:** Checks if the sign-in button is rendered when the session is null. It mocks the session status and checks for the presence of the sign-in button.
- **calls sign in function when sign in button is clicked:** Tests if the sign-in function is called when the sign-in button is clicked. It simulates a button click event and asserts that the `signIn` function is called.

repoComponents.test.js

- **renders the component:** Verifies that various repository-related components render correctly. Each test ensures that a specific component renders the expected text or elements.

RepoPage.test.js

- **renders correctly:** Tests whether the RepoPage component renders correctly. It mocks the session and fetches data for a specific repository, then checks if the repository name is displayed in the rendered component.

UserComparePage.test.js

- **renders correctly:** Ensures that the UserComparePage component renders correctly. It mocks the session and fetches data for two users to compare, then verifies the presence of specific text indicating user comparison.

userComponents.test.js

- **renders the component:** Tests whether various user-related components render correctly. Each test checks if a specific component renders expected text or elements.

UserPage.test.js

- **renders correctly:** Verifies that the UserPage component renders correctly. It mocks the session and fetches user data, then checks if the user's name is displayed in the rendered component.

Backend Integration Testing with Pytest (API Endpoints)

The following will outline the integration testing strategy for the backend API endpoints of Gitulyse using pytest. Integration tests aim to verify that the different parts of the system work together correctly. The backend API endpoints provide functionality related to repositories, commits, pull requests, issues, and user activity on GitHub. Note that we achieved 100% coverage on all of our backend API files.

Integration Test Coverage

Below is a coverage report for our Integration tests:

<i>Module</i> ↑	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>branches</i>	<i>partial</i>	<i>coverage</i>
gitulyse_api__init__.py	55	0	0	16	0	100%
gitulyse_api\commits.py	44	0	0	18	0	100%
gitulyse_api\db.py	14	1	0	6	1	90%
gitulyse_api\issues.py	66	0	0	22	0	100%
gitulyse_api\pull_requests.py	63	0	0	20	0	100%
gitulyse_api\repos.py	62	0	0	18	0	100%
gitulyse_api\search.py	31	0	0	10	0	100%
gitulyse_api\users.py	84	0	0	36	0	100%
Total	419	1	0	146	1	100%

- test_commits.py: Tests related to commit statistics retrieval.
- test_db.py: Tests for database initialisation and teardown.
- test_factory.py: Tests related to app factory creation, index endpoint, and activity statistic retrieval.
- test_issues.py: Tests for retrieving and analysing issues.
- test_pull_requests.py: Tests for pull request retrieval and analysis.
- test_repos.py: Tests for repository-related functionalities.
- test_search.py: Tests for search functionality.

Test Environment Setup

The test environment is set up using fixtures provided by Pytest. These fixtures include mocking database connections, GitHub API responses, and giving client instances for testing.

```
@pytest.fixture
def app(mock_db):
    app = create_app({"TESTING": True, "MOCK_DB": mock_db})

    with app.app_context():
        get_db()

    yield app
```

GoCodeo-Generate tests for the below function

```
@pytest.fixture
def client(app):
    return app.test_client()
```

GoCodeo-Generate tests for the below function

```
@pytest.fixture
def runner(app):
    return app.test_cli_runner()
```

GoCodeo-Generate tests for the below function

```
@pytest.fixture
def mock_db():
    return mongomock.MongoClient()
```

Test Cases

1. test_commits.py

- test_code_contribution_stats: Verifies the correctness of code contribution statistics retrieval.
- test_parse_contributions_single_user: Tests parsing contributions for a single user.

2. test_db.py

- test_get_db: Ensures that the database is correctly initialised.
- test_close_db: Checks if the database connection is properly closed.
- test_init_app: Verifies if the teardown function is registered correctly.

3. test_factory.py

- test_create_app_config: Checks app configuration based on input.
- test_index: Ensures the index endpoint returns the expected response.
- test_github_activity: Verifies GitHub activity endpoint functionality.
- test_github_activity_day: Tests GitHub activity endpoint for a specific day.

4. test_issues.py

- test_get_issues: Tests issue retrieval and analysis.
- test_get_percentage_issues: Verifies percentage of resolved issues calculation.
- test_get_percentage_issues_no_issues: Ensures correct behaviour when no issues are present.
- test_get_percentage_issues_incorrect_dates: Checks handling of incorrect date inputs.

5. test_pull_requests.py

- test_get_pull_requests: Verifies pull request retrieval functionality.
- test_get_percentage_pull_requests: Tests percentage of merged pull requests calculation.
- test_get_percentage_pull_requests_no_pull_requests: Ensures correct behaviour when no pull requests are present.
- test_get_percentage_pull_requests_incorrect_dates: Checks handling of incorrect date inputs.

6. test_repos.py

- test_get_repos: Tests repository retrieval functionality.
- test_get_repos_incorrect_token: Ensures invalid token handling.
- test_get_repos_limit_one: Verifies limiting the number of repositories returned.
- test_get_repos_force_true: Tests forcing repository retrieval.
- test_get_repos_force_true_extra_repo: Verifies forcing repository retrieval with extra repos.
- test_get_repos_last_update_doc_not_none: Checks handling of non-None last update document.
- test_get_repos_last_update_doc_not_none_time_diff: Ensures correct behaviour with the time difference in the previous update document.
- test_get_repos_no_commits: Verifies behaviour when no commits are present.
- test_get_repo_stats: Tests repository statistics retrieval.
- test_get_repo_stats_incorrect_token: Ensures invalid token handling for repository statistics.

7. test_search.py

- test_search_no_query: Verifies handling of missing query parameter.
- test_search_no_type: Ensures handling of missing type parameters.
- test_search_user: Tests user search functionality.
- test_search_repo: Verifies repository search functionality.
- test_search_invalid_credentials: Ensures invalid token handling during the search.
- test_search_index_error_users: Checks handling of index errors during user search.
- test_search_index_error_repos: Verifies handling of index errors during repository search.

8. test_users.py

- test_get_user: Verifies the successful retrieval of user data from the Gitulyse API endpoint.
- test_get_user_cached_data: Tests the caching mechanism by ensuring that cached user data is returned when the same user requests data again without forcing a refresh.
- test_get_user_force: Validates the ability to force data refresh by setting the 'force' parameter to 'true'.
- test_get_user_no_stats_contributors: Ensures handling of cases where user repositories do not have statistics contributors data available.
- test_get_user_no_user_given: Tests the scenario where no specific user is provided in the API request. It confirms that the API responds with user data for a predefined default user.
- test_get_user_no_user_found: Validates the handling of cases where the requested user is not found.
- test_get_user_incorrect_token: Verifies the response when an invalid user token is provided.

Conclusion


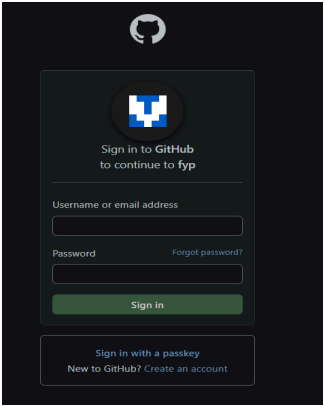
The integration tests comprehensively cover the backend API endpoints of Gitulyse, ensuring the functionality works as expected under various scenarios. These tests provide confidence in the stability and correctness of the backend API.

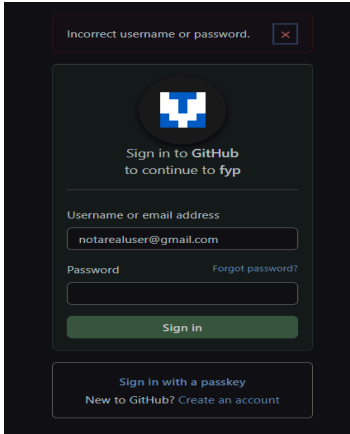

Ad-Hoc Testing

Ad-hoc testing for Gitulyse involves exploring the application in an unscripted and exploratory manner to uncover usability issues, unexpected behaviours, and edge cases that may not be covered by automated tests. Testers engage with the application organically, interacting with different features, inputs, and workflows to identify any anomalies or areas for improvement. Ad-hoc testing encompasses a wide range of scenarios, including user interface layout, responsiveness, error handling, and performance under various conditions. This iterative and dynamic approach complements automated testing efforts, contributing to the overall reliability and usability of the application.

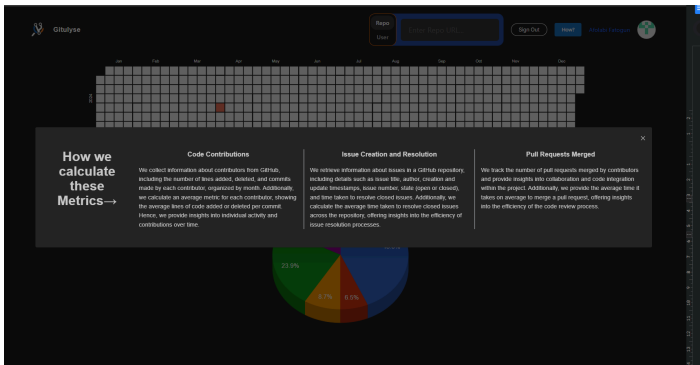
For Ad-hoc we followed the following testing document

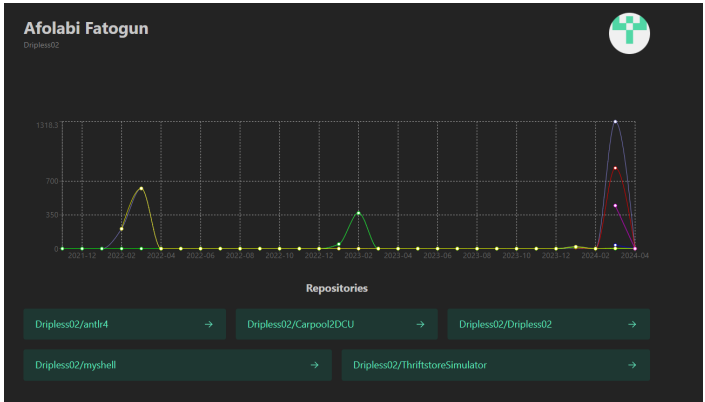
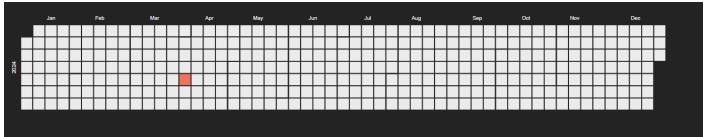
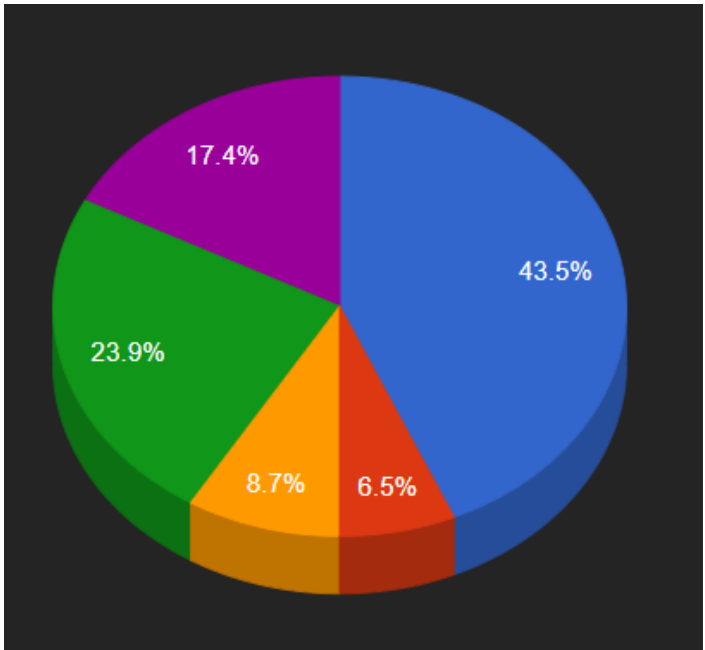
Login Function Testing

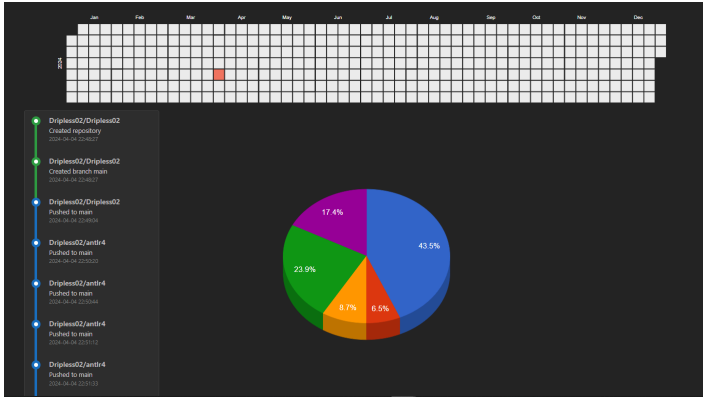
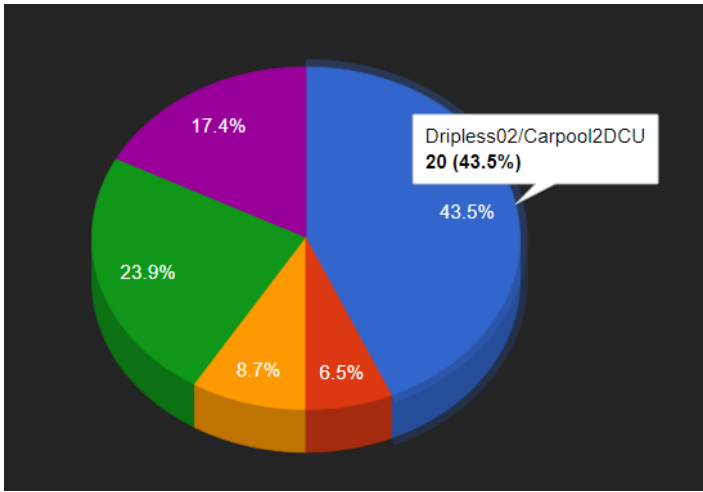
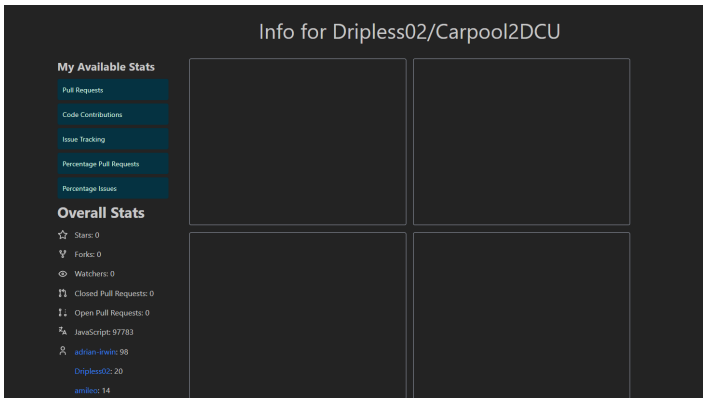

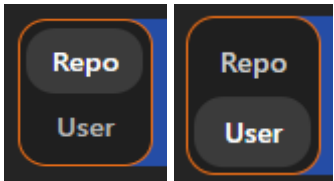
Action	Result	Expected Result	Evidence
Click the 'Login button'	Taken to sign in with the GitHub page	✓	
Click the 'Sign in with GitHub' button	Taken to form where you can sign in with your GitHub account	✓	

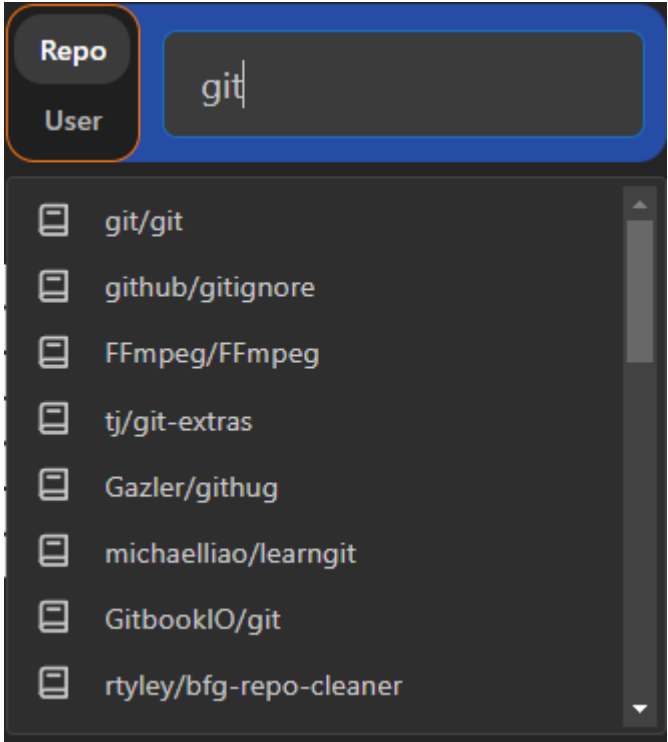
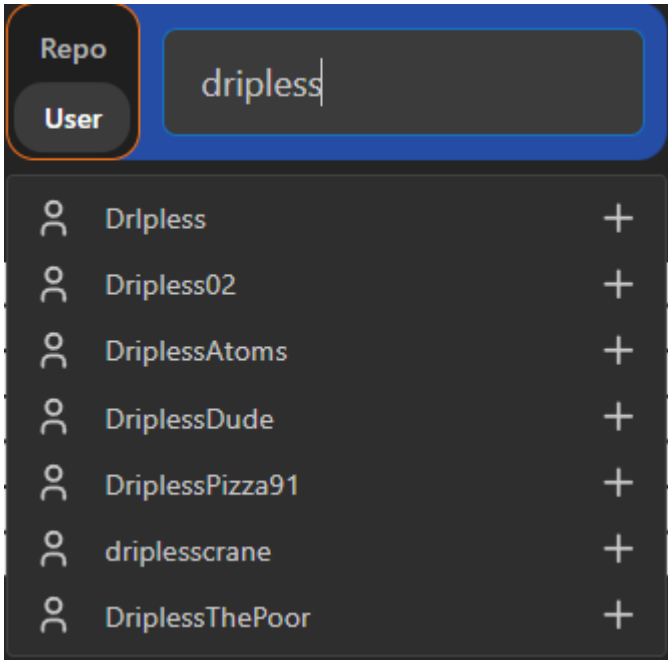
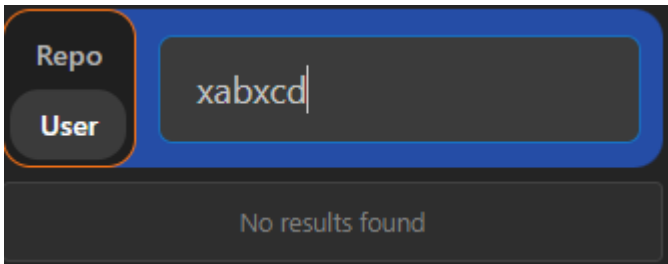
Enter Invalid GitHub credentials	Incorrect username or password message is displayed	✓	
Enter Valid GitHub account details	Relocated to the home page and can see your username and profile picture at the top right of the screen	✓	
Click Sign Out	Go back to the sign-in page	✓	N/A

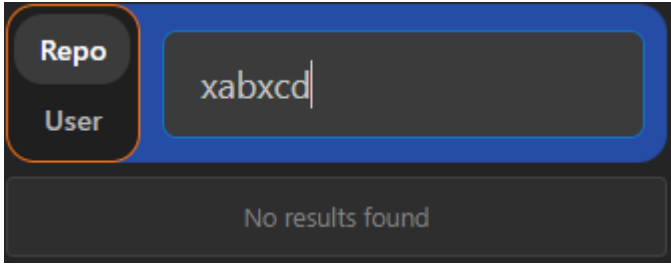
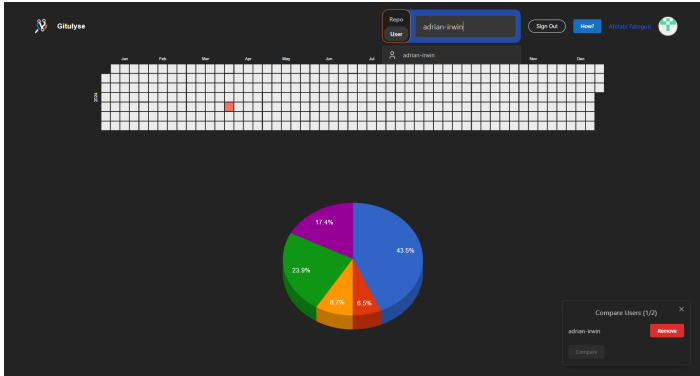
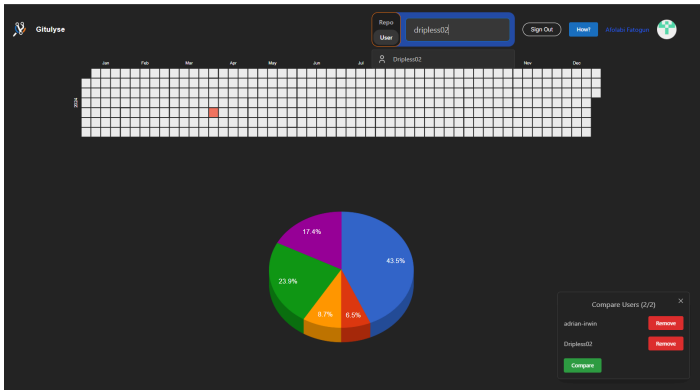
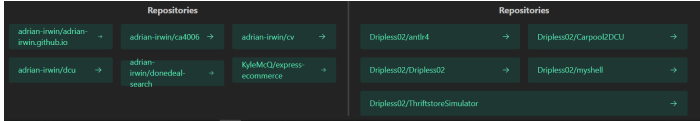
Home Page/ Nav Testing

Action	Result	Expected Result	Evidence
Click on the Homepage/Logo button	Nothing should happen as we are already on the homepage	✓	N/A
Click on the How? Button on the Nav bar	A modal containing information on how the metrics are calculated is displayed	✓	

While the modal is active click anywhere on the screen outside of the modal	Modal disappears	✓	N/A
Click on their Github User Icon	<p>A loading screen should appear</p> <p>Once the loading screen fades, the user sees a graph with the contributions to all their repos since their account was active,</p> <p>The User can see their repositories</p>	✓	
Verify that the calendar is rendered	Calendar is displayed with a user activity for the current year	✓	
The user verifies that the Pie chart is rendered with percentages indicating their 6 most active repos	Users can see the Pie chart	✓	

Click an active day on the calendar	A log of the user's activity for that day is displayed	✓	
Hover on the Pie chart sections	A tooltip with the Repo name, number of commits, and the percentage of work out of a user's total is displayed	✓	
Click on any of the Pie chart sections	The user is redirected to an information page for that repo	✓	
Verify the search bar component	The user can see the search bar	✓	
Click the Repo and User button on the search bar component	The user can select the Repo and User options on the search bar	✓	

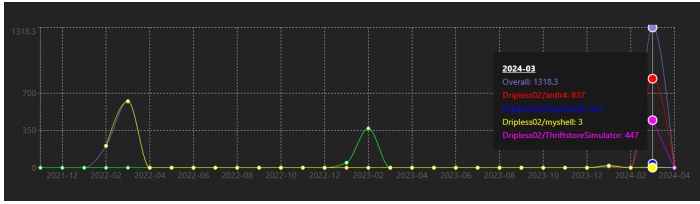
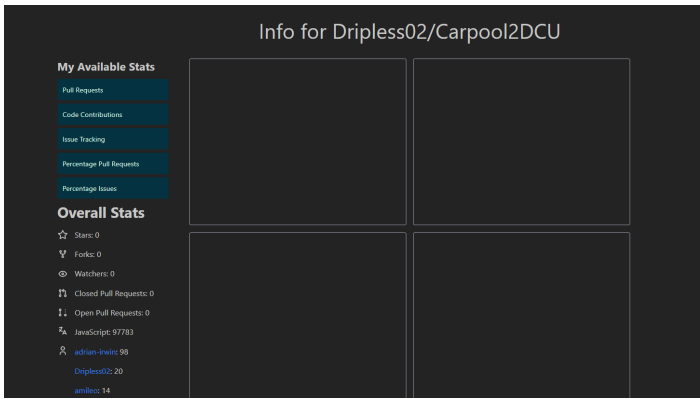
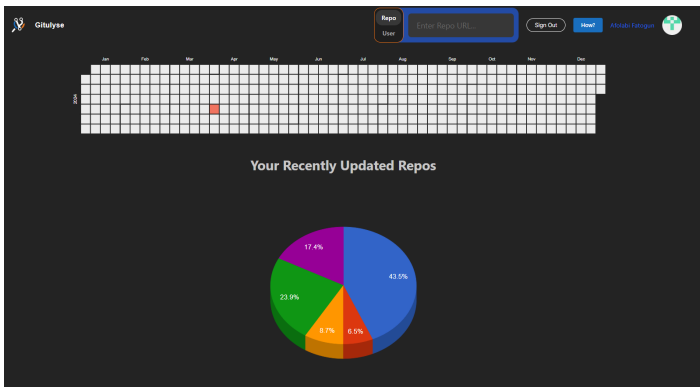
Start searching for a repo name in the repo section i.e. "git"	Users can see pop-up suggestions while searching for a repo	✓	 <p>The screenshot shows a search bar with 'git' entered. The 'Repo' tab is selected. Below the search bar, a list of repository suggestions is displayed, each with a repository icon and the name: git/git, github/gitignore, FFmpeg/FFmpeg, tj/git-extras, Gazler/github, michaeliao/learngit, GitbookIO/git, and rtyley/bfg-repo-cleaner.</p>
Start searching username in the user section i.e. your own user name	Users can see pop-up suggestions while searching for a user	✓	 <p>The screenshot shows a search bar with 'dripless' entered. The 'User' tab is selected. Below the search bar, a list of user suggestions is displayed, each with a user icon and the name: Dripless, Dripless02, DriplessAtoms, DriplessDude, DriplessPizza91, driplesscrane, and DriplessThePoor. Each suggestion has a '+' icon to its right.</p>
Search for a user that doesn't exist i.e 'xabxcd'	Pop-up returns 'no results found'	✓	 <p>The screenshot shows a search bar with 'xabxcd' entered. The 'User' tab is selected. Below the search bar, the text 'No results found' is displayed.</p>

Search for a repo that doesn't exist i.e 'xabxcd'	Pop-up returns 'no results found'	✓	
Click the plus button beside a user's name in the user search	A new pop-up to compare users appears	✓	
Search for a second user and click the plus button beside their name	Their name is added to the compare list in the popup	✓	
Click the remove button in the popup beside one of the user's names	The user is removed from the compare list	✓	N/A
Click the Compare button after two users have been added to the Compare list	A loading screen appears followed by a side-by-side comparison of the users.	✓	

--	--	--	--

User Page Testing

Action	Result	Expected Result	Evidence
Verify that the following components are on the user page: - User's Full name - username - Timeline graph - Repositories	All components are present on the page	✓	

Hover on the timeline graph	Users can see a tooltip with the user's contributions to their repos within that time frame	✓	
Click on a repository link	The user is redirected to the information page of the repository	✓	
Click on the Home/Logo button	The user is redirected to the homepage	✓	


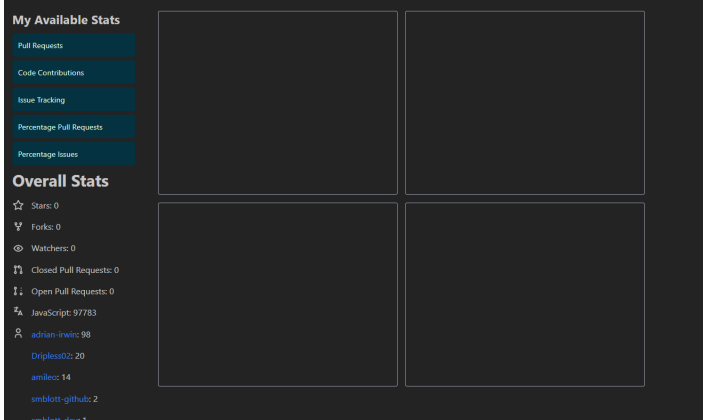

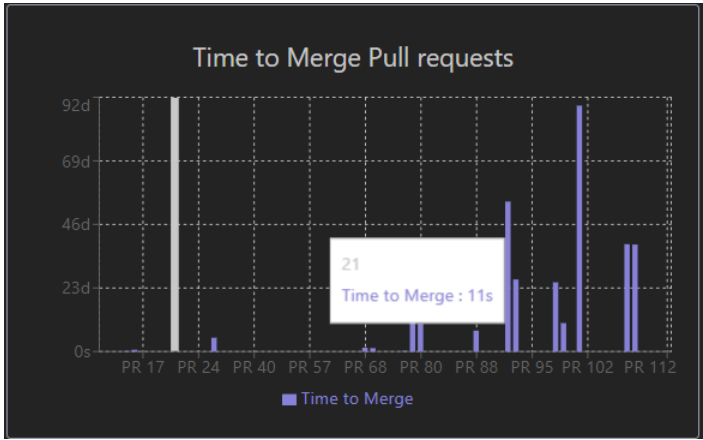

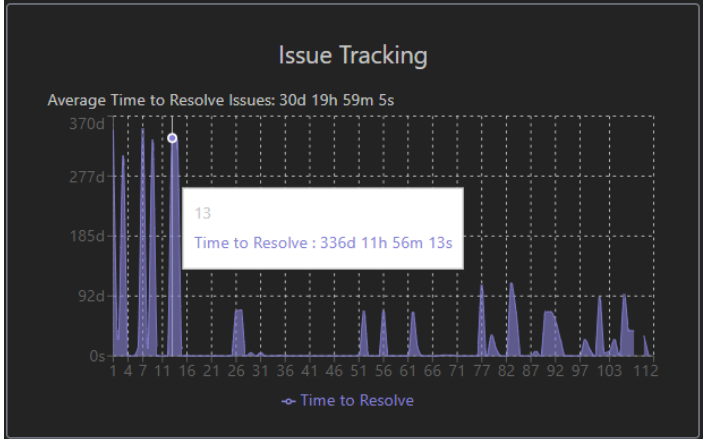
Compare Page Testing

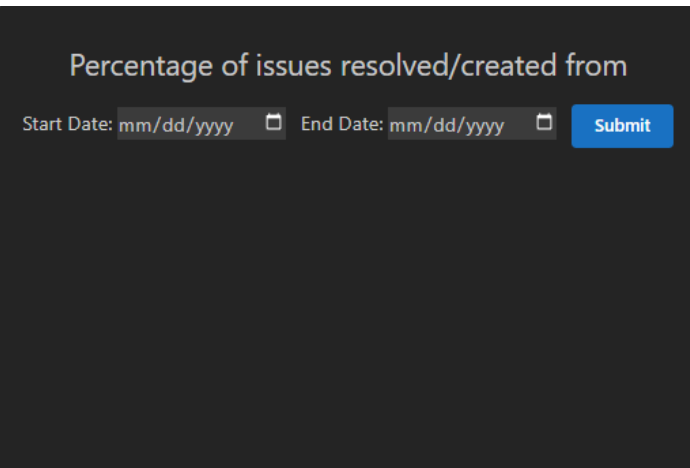
Action	Result	Expected Result	Evidence
<p>Verify the following components on the compare page:</p> <ul style="list-style-type: none"> - 2 users - 2 user timeline graphs - 2 user profile 	All components are present on the page	✓	

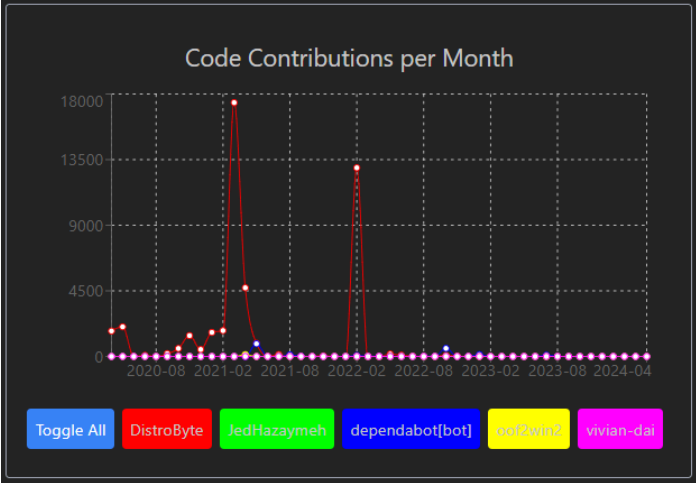
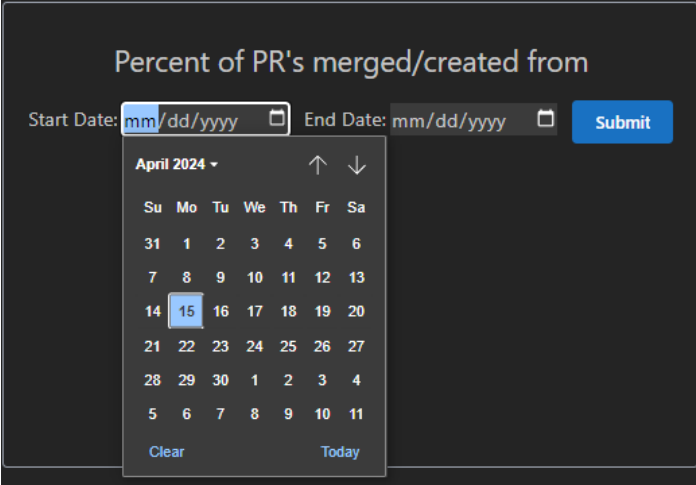
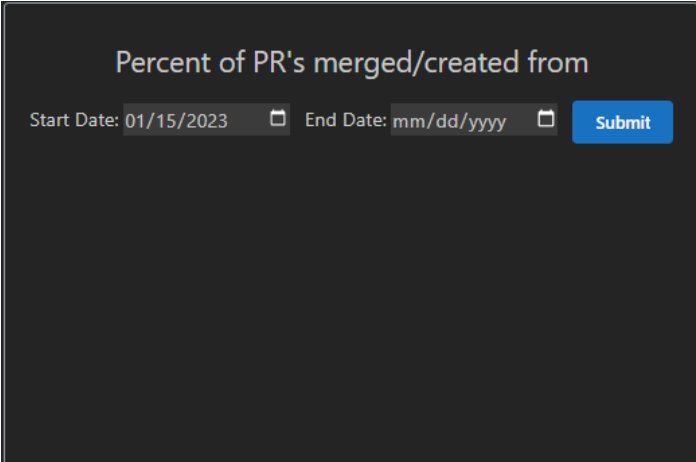
<p>pictures</p> <p>- Account statistics</p> <p>-Repositories</p>			
<p>Hover on the timeline graph</p>	<p>Users can see a tooltip with the user's contributions to their repos within that time frame</p>	✓	
<p>Click on a repository link</p>	<p>The user is redirected to the information page of the repository</p>	✓	
<p>Click on the Home/Logo button</p>	<p>The user is redirected to the homepage</p>	✓	

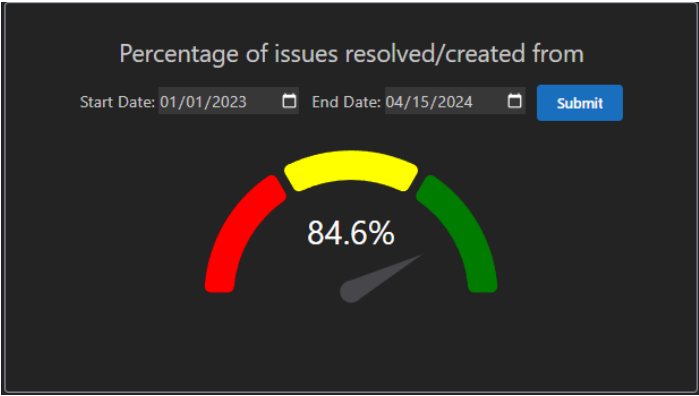
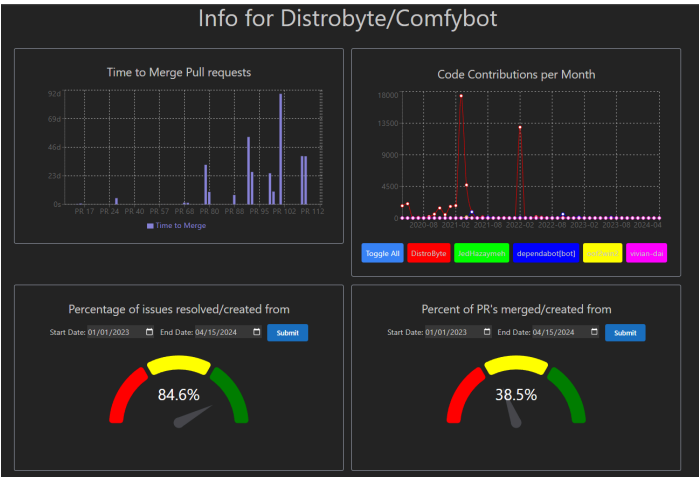
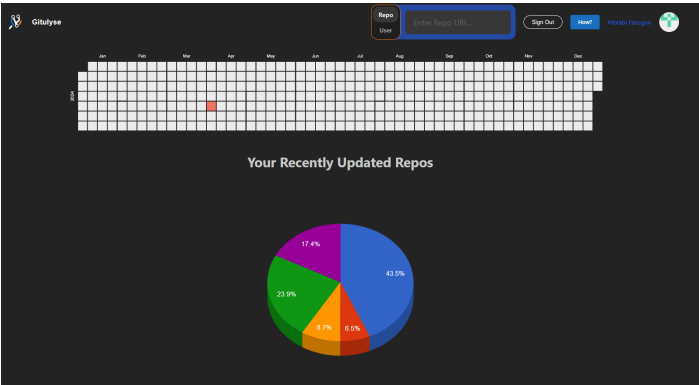
Info Page Testing

Action	Result	Expected Result	Evidence
--------	--------	-----------------	----------

<p>Verify the following components on the Info page:</p> <ul style="list-style-type: none"> - Available Stats: <ul style="list-style-type: none"> - Pull requests - Code Contributions - Issue tracking - Percentage Pull requests - Percentage Issues - Overall Stats - 4 Drop Points 	<p>All components are on the page as expected</p>		
<p>Drag PullRequests stat from 'My Available Stats' to Drop point</p>	<p>Pull request statistic is rendered in a drop point</p>		
<p>Drag Issue tracking stat from 'My Available Stats' to Drop point</p>	<p>Issue tracking statistic is rendered in a drop point</p>		

Drag Code contribution stat from 'My Available Stats' to Drop point	Code contributions statistic is rendered in a drop point	✓	
Drag percentage pull request stat from 'My Available Stats' to Drop point	Percentage pull request statistic is rendered in a drop point	✓	
Drag percentage issues stat from 'My Available Stats' to Drop point	Percentage issues statistic is rendered in a drop point	✓	

Click toggle all on a Code contributions per month stat	All author contributions are rendered at the same time without crashing/lagging	✓	
Click to edit the Start/End date of the percentage stats	The date popup is displayed	✓	
Enter a valid date into the section	The valid date is recorded in the section	✓	

Enter two valid dates and click submit	The gauge of the percentage of PRs merged/resolved issues is displayed	✓	
Fill all 4 Drop points with Stats at the same time	All 4 components are rendered with no issues	✓	
Edit stats with configurable options while all other drop points are filled	All other stats are unaffected by editing configurable stats	✓	N/A
Click the Home/Icon button	The user is navigated to the homepage	✓	

User Testing

User testing verifies the usability and relevance of the system from end-to-end users. The user testing phase typically involves a group of participants representing Gitulyse's target user demographics. This ensures that the platform caters to a wide range of users with varying levels of technical expertise, preferences, and requirements. By including users with different backgrounds and skill sets, Gitulyse gains valuable insights into how various individuals interact with the platform and can tailor its features and interface accordingly.

In addition to task-based testing, we also solicited feedback from participants regarding their overall experience with the platform using a survey. Participants are encouraged to voice their opinions, provide suggestions for improvement, and highlight any pain points they encountered during the testing process. This feedback is invaluable for identifying areas where the platform excels and where it can be enhanced to better meet user needs. Below is the user survey we would have used in validating the system.

Onboarding Experience

⌵⋮

Description (optional)

How easy was it for you to connect your github with Gitulyse? *

12345

Not easy☐ ☐ ☐ ☐ ☐ Very easy

Were you able to connect your Git repositories without any issues? *

☐ Yes

☐ No

How intuitive do you find the user interface of Gitulyse? *

Easy 1 2 3 4 5 Difficult

☐ ☐ ☐ ☐ ☐

Did you encounter any difficulties in finding specific functionalities within Gitulyse? *

- ☐ Yes
- ☐ No
- ☐ Maybe

If you answered yes to the previous, please describe the issue, if No input N/A. *

Long-answer text

Feature Usefulness



Description (optional)

Which features of Gitulyse have you found most useful in managing your Git repositories? *

- ☐ User Comparison Feature
- ☐ Repository Information Feature

How often do you utilize the code analysis and visualisation tools provided by Gitulyse? *

	1	2	3	4	5	
Not at all	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Very often

Are there any features you expected to find in Gitulyse but couldn't locate? *

- ☐ Yes
- ☐ No

If you answered yes to the previous, describe the feature, if No input N/A. *

Long-answer text

Performance and Reliability



Description (optional)

Have you experienced any performance issues while using Gitulyse (e.g., slow loading times, lagging)? *

☐ Yes

☐ No

Have you encountered any bugs or technical glitches while using Gitulyse? *

☐ Yes

☐ No

If you answered yes to the previous, describe bug or technical glitch encountered, if No input N/A. *

Long-answer text

Overall Satisfaction and Recommendations

Description (optional)

On a scale of 1 to 5, how satisfied are you with your experience using Gitulyse? *

	1	2	3	4	5	
Not satisfied	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Very satisfied

Would you recommend Gitulyse to colleagues or peers working in software development? *

☐ Yes

☐ No

Future Work

Through testing and feedback, Gitulyse has proved a powerful tool for analysing GitHub repositories, providing insights into code contributions, issue tracking, pull requests, and user activity. However, to remain relevant in the ever-changing landscape of software development, Gitulyse must evolve and adapt. The following will explore potential future integration paths that can further enhance Gitulyse's analytical capabilities and user experience. By leveraging other technologies and best practices, Gitulyse can continue to empower developers, project managers, and researchers to gain valuable insights from their GitHub data.

Continuous Integration and Deployment (CI/CD) Integration:

Integration with CI/CD platforms such as Jenkins, Travis CI, or GitHub Actions can further enhance Gitulyse's capabilities by providing insights into the continuous integration and deployment process. By analysing build logs, test results, and deployment pipelines, Gitulyse can help developers identify performance bottlenecks, detect failures early, and optimise their CI/CD workflows for better efficiency and reliability. Additionally, integration with CI/CD platforms can enable Gitulyse to track the impact of code changes on build and deployment metrics, providing valuable feedback to developers and project managers.

Machine Learning and Predictive Analytics:

Another promising direction for Gitulyse is the integration of machine learning and predictive analytics capabilities. By leveraging historical GitHub data, Gitulyse can train machine learning models to predict future trends in code contributions, issue resolution times, and summarise Readme/documentation files. These predictive insights can help developers and project managers make informed decisions and proactively address potential issues before they arise. Additionally, machine learning algorithms can be used to identify patterns and anomalies in GitHub data, enabling users to gain deeper insights into their projects' dynamics.

Conclusion

By exploring future integration paths such as enhanced data visualisation, machine learning, CI/CD integration, and accessibility, we can get even more efficient as a GitHub analytics tool.