FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Metrics of software development workflow
Student:	Basel Samy Mohamed Kamaleldin Elshanawany
Supervisor:	Ing. Tomáš Vondra, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

The company Pure Storage's cloud division uses the GitHub platform for software development projects. They would like to collect some metrics from that system.

1. Analyze their workflow and how the platform supports it. Study means of automated data collection from GitHub web API. Traverse the tree of projects, commits, etc. and collect data into a relational database.

2. Design an application to collect, store, and visualize metrics. Some suggested metrics are Code Coverage, Pull Request wait time for review, number of PRs per version, etc. It should be possible to display a dashboard of different repositories based on the project, department etc. with calculated statistics and see the history of the individual metrics with AI or statistical extrapolation.

3. Implement the application in the Python language with correct object-oriented design.

4. Test it on a big open-source project (such as Docker, Kubernetes) or a mock project that mimics the company workflow (will be provided).

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 7 November 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Metrics of Software Development Workflow

Basel Samy ElShanawany

Department of Theoretical Computer Science
Supervisor: Ing. Tomáš Vondra, PhD

January 6, 2022

Acknowledgements

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on January 6, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Samy ElShanawany, Basel. *Metrics of Software Development Workflow.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Cílem této práce je analyzovat metriky pracovního toku vývoje softwaru a vytvořit webovou aplikaci, která pomáhá vývojářům softwaru vizualizovat a sledovat jejich produktivitu. Analýza vedla ke shromažďování dat z určitých koncových bodů rozhraní API GitHubu prostřednictvím zprostředkovatele zpráv hostovaného na Heroku a k vytvoření fronty úloh pro asynchronní volání a ukládání dat z API mimo hlavního cyklu žádost-odpověď webové aplikace. Strana serveru byla vyvinuta v Pythonu s webovým frameworkem Django, klientská strana komunikuje s REST API Django a integruje se s Plotly Dash, což umožňuje vytváření interaktivních vizualizací

Klíčová slova Doba cyklu, Django, GitHub API, Architektura MVC, Python, REST API, webová aplikace

Abstract

The goal of this thesis is to analyze software development workflow metrics and build a web application that helps software developers visualize and track their productivity. The analysis lead to data collection from certain endpoints of GitHub's API through a message broker hosted on Heroku and a task queue worker to asynchronously call and store data from the API outside of the

web application's request-response cycle time. The server side was developed in Python with Django's web framework, the client side communicates with Django's REST API and integrates with Plotly Dash which allows for the creation of interactive dashboard visualizations

Keywords Cycle Time, Django, GitHub API, MVC Architecture, Python, REST API, Web Application

Contents

List of Figures

List of Listings

Introduction

In most industries there are established ways of measuring how well an organization is doing; for example in the sales industry, having more sales would directly indicate that an organization is healthy and productive. However, in the software development industry it's more of a challenge to measure productivity as there's no established ways that indicate how well a software product is doing during its development phase

Software Development Methodologies

There are many software development methodologies that are used by teams to structure out and tackle software projects. The "Waterfall" methodology is a traditional, sequential method meaning it's initially very structured in the form of requirements which are then developed very rigidly in stages allowing for very minimal flexibility; this methodology is now a days seen as an outdated "oldschool" method. Alternatively, methodologies such as "Agile" or "SCRUM" allow for a more flexible, collaborative environment where teams develop in short "sprints" (iterations) each of which has a list of deliverable outcomes [1]. Agile methodology claims to increase productivity, however, there are no metrics that are explicitly defined or commonly agreed upon

Software Companies that follow an Agile methodology usually have an automated Continuous Integration (CI) / Continuous Deployment (CD) pipeline which allows them to automate the software delivery process and induces a more rapid development environment

Outline

The first chapter focuses on finding a suitable metric to measure software productivity as well as the integrated systems within the CI / CD pipeline

where the data is going to be collected from, afterwards the design phase of the web application begins where the main structure of the application is outlined, a suitable data model is devised and a user interface is designed. Then the implementation phase where the chosen data is collected from the CI / CD pipeline and the graphical visualisations are built to visualise the metric in a useful way

Chapter 1

Analysis

1.1 Software Development Workflow Metrics

As previously mentioned productivity can be tricky to measure in software development as the "output" can't be measured accurately; metrics can also be a point of controversy as poorly chosen metrics or the way in which they are presented can incentivize bad habits within a team

1.1.1 Existing Metrics

By researching existing metrics that are used within software development teams to measure productivity I was able to gain a better understanding of the areas which are useful to measure as well as identify the points of strength and weaknesses of these metrics in order to then apply that knowledge towards my devised metric

1.1.1.1 Commits

This metric is based on the number of commits being pushed to a GitHub repository. It is a poor measurement of output as the number of commits doesn't equally translate into the value of the additions as well as the quality of the code. However, it can be useful for incentivizing a good habit of having small, frequent commits which allows for greater transparency, collaboration and continuous integration

1.1.1.2 Lines of Code

This metric is based on the lines of code being added to a GitHub repository. It is a poor measurement of output as there's differences in languages and the way code is formatted, also, having more line of code is not necessarily a good thing usually when refactoring code you are trying to achieve the same result

while writing less code. It can however be a useful metric for understanding the size of a software system and how a code base is changing

1.1.1.3 Pull Request Count

This metric is based on the number of pull requests being added to a GitHub repository. It is a poor measurement of output as it doesn't take into account the amount of effort or difficulty put into the work as well as its value, the way this metric is used by a team can also encourage unnecessarily small pull requests. It can however be a useful metric for understanding the release times in a CD pipeline

1.1.1.4 Velocity Points

This metric is often used in Agile software development where the main idea is to help teams estimate how much work they can complete based on how quickly similar work was previously completed [2]. The main two elements of the metric are the "units" chosen by the team to measure velocity such as engineer-hours or "story points" as well as the duration of the iteration most often weekly but in some cases monthly. It is a poor measurement of output as the sizing (units) are estimated before the work is completed and not after, therefore, it could incentivize teams to inflate their velocity score by over estimating the time it takes to complete a task [3]. It can how ever be a useful metric when not viewed as a measure of team performance and rather for understanding delivery forecasts based on past estimates

1.1.1.5 Impact

This metric is based on how many files were modified / newly added as well as how much code was modified / newly added which is then accumulated to calculate an "impact" score. It has the same flaws as the lines of code and is too abstract to be actionable

1.1.2 Why Measure Productivity

After analysing some of the existing metrics and understanding why some of them were not able to capture productivity; it prompted the question of why should we measure productivity, by identifying these reasons a suitable metric can then be determined to align with these reasons

1.1.2.1 Prompt Action

Metrics can help justify certain actions that need to be taken to better improve the workload across a team or shed light on some issues that might have been overseen; for example the assignment of code reviews could be gathered across

a team to determine in real-time who should be assigned to the next upcoming code review or for open source maintainers to help diagnose issues / reviews that have been opened for a long time and need to be followed up on

1.1.2.2 Goals & Alignment

A more essential reason for measuring productivity is to quantitatively express a desired goal and verify whether you are actively achieving it or to analyze previous stages which where successful to advocate certain changes to be made

1.1.2.3 Advocation

On a business level a metric could help provide a valid reasoning for any changes or additions to be made to a team that could then in turn help improve the productivity and workflow

1.1.3 What to Measure

An important aspect for devising a valuable metric is in first identifying what areas to measure within a workflow that would be beneficial towards improving it, as well as the ways to contextualise the metric data for it to be presented in a useful way

1.1.3.1 Process vs Output

Measuring the "output" of a software project is not justifiable as seen in the previously mentioned metrics as there usually isn't a direct correlation between the "output" and the productivity. Any high performing software organization is not one that necessarily has a high "output" but rather one than is responsive to one another and collaborates efficiently. Therefore by measuring a process within an organization and improving it, it directly impacts the day to day experience of a developer and the overall workflow. On a managerial level having metrics that measure the process productivity of teams can help them quantitatively express improvements to their teams

1.1.3.2 Measuring Against Set Targets

Visualising the absolutes or true values of a process doesn't give much insight as to how well you are doing and can also incentivize bad habits. Therefore a better approach is to measure against targets that are self defined which also allows for flexibility of target goals across different teams and scopes

1.1.4 Cycle Time Metric

Cycle Time can be seen as a measure of process speed, it's a metric borrowed from lean thinking and manufacturing disciplines [4]. The process' beginning

time can be defined by the time the ticket is issued or by the time of the first commit made to GitHub. The end time is when the production code is deployed to the end users. Within that larger process time lies other smaller cycle times such as the coding time, review time and deploy time as seen in Figure 1.1. The average cycle time can say a lot about a team's software development practices and the tools used throughout their CI / CD pipeline such as their code review tools, automated tests and deployment scripts. It can help a team quickly identify and diagnose long wait times or bottlenecks within their pipelines

1.1.4.1 Coding Time

Is the time between the first commit being pushed to a given branch and the moment a pull request is created for that branch. As each smaller cycle feeds into the larger overall cycle time, by monitoring the coding time it encourages dividing the work into smaller more manageable chunks which would then improve the overall cycle time

1.1.4.2 Review Time

Is the time between the pull request being issued and the pull request being merged into the master branch. By monitoring this metric it helps teams process code reviews in a timely manner and helps prevent large pull requests that are too large to review effectively

1.1.4.3 Deploy Time

Is the time between the pull request being merged into the master branch and the code being deployed into production. By monitoring this metric it helps teams that track the deployment time identify certain bottlenecks and improve streamlining builds and automated deployments

1.1.5 Chosen Metric

As per the analysis made on the existing metrics, why we should measure productivity and the appropriate way of how measuring productivity should be approached. I decided on capturing two measurements of productivity focusing one on a process and one on an "output". The process I would like to measure is the review time which lies within the cycle time. As for the output it would measure the size of pull requests that are being merged into a repository

1.1.5.1 Pull Request Wait Time

The review time metric is essentially a sub process within the larger process which is the overall "cycle time" that measures the time it takes from when a

task is issued to when its deployed into production. The review time focuses on the process of when the coding is already done and the pull request is issued on GitHub, the time it takes for a reviewer to thoroughly review the code, conduct appropriate tests and then merge the code into the master branch for it to be released. The way of visualising this metric in a useful way may not be to show the absolute values of how long the review times take but rather to allow teams to setup personal goals and display a success rate of how well the are doing with respect to their set goal. I choose to measure this stage of the cycle time process as it's the mid point in the code's life cycle where the logic of the code is being validated as well as its readability and quality. This would help teams better identify problems within their continuous integration pipeline such as the automated testing and better diagnose them to speed up this process and in turn their overall "cycle time"

1.1.5.2 Pull Request Size

The pull request size metric is a measure of "output" in a sense where the measurement is the number of additions and deletions being merged into the master branch to be deployed into production. This metric doesn't suffer from the same flaws as the lines of code metric as it only takes into account code that has been through the review process and is then successfully merged. It can be useful for understanding the size of your software system as well as how the code base is changing, it also is a good indicator for better estimations of release times within a CD pipeline. By viewing this metric you can also identify whether you are dividing the tasks into suitable manageable chunks of work which would in turn help improve your coding time as well as review time and therefore your overall "cycle time"

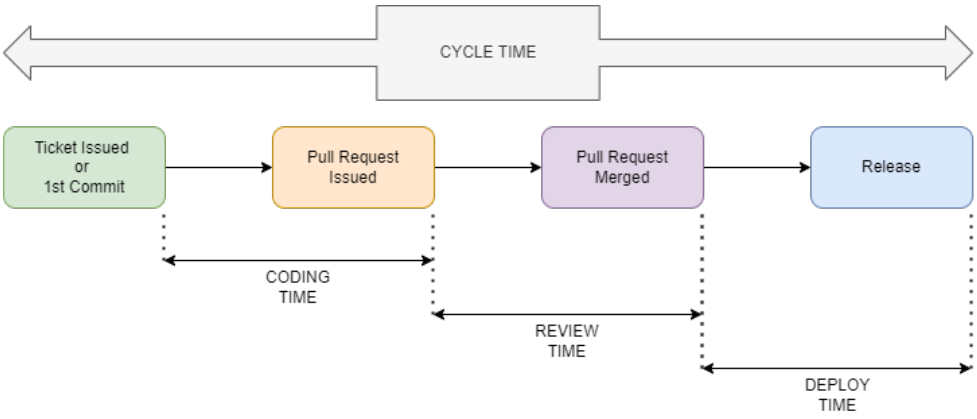

Figure 1.1: Cycle Time Metric in Software Development Workflow

1.2 DevOps

DevOps workflow focuses on bridging the gap between the development and operations teams by:

- Continuous Development / Integration (CI)
- Continuous Monitoring / Feedback (CM)
- Continuous Delivery / Deployment (CD)

1.2.1 GitHub

GitHub is a powerful Version Control System (VCS) that allows developers to keep track of revisions (versions) and changes in a projects code; it also allows for a pipeline that automates the software delivery process

1.2.1.1 Version Control Systems (VCS)

Using VCS such as GitHub is a great practice within a high performing software and DevOps team as it reduces development time and increases the rate of successful deployments. Essentially, VCS allow for a logically structured way to view changes made across the source code in a "file tree" structure while supporting multiple revisions of the same source code to coexist simultaneously allowing for different developers working on separate "branches" not to be affected by each other changes. The merging of features (branches) could later result in conflicts or bugs within the code where the VCS provides a mechanism for reverting the source code back into a previous revision. This structured overview that VCS provide is essential to the DevOps development environment as it eases the process of maintaining the code during its development, bug tracking as well as the deployment of the code into production

1.2.1.2 GitHub's Structure

GitHub provides "organization" accounts which allows team members to collaborate on several shared projects. Owners and managers can manage each member's access through security and administrative features [5] which allows an organization to emulate the real life schema of the projects being developed at a company and the development teams working on those projects. Each project would have its relevant code base (repository) setup within the organization account where the developers who are working on that project have read and write access

1.2.1.3 GitHub's Flow

"Gitflow" is a branching model for Git, created by Vincent Driessen [6]. The main idea behind this workflow is that the master branch tracks only released

code then there's a main development branch which handles all the constant integration for new feature branches. When it's time for deployment a release branch is based off of the development branch. This code is then tested thoroughly and any changes or bug fixes are made directly on the release branch which can then be merged into the master branch to be deployed into production as well as back into the development branch to ensure any changes made are maintained throughout the next development cycle. This workflow allows for ease of parallel development and collaboration as well as a release staging area for the development branch

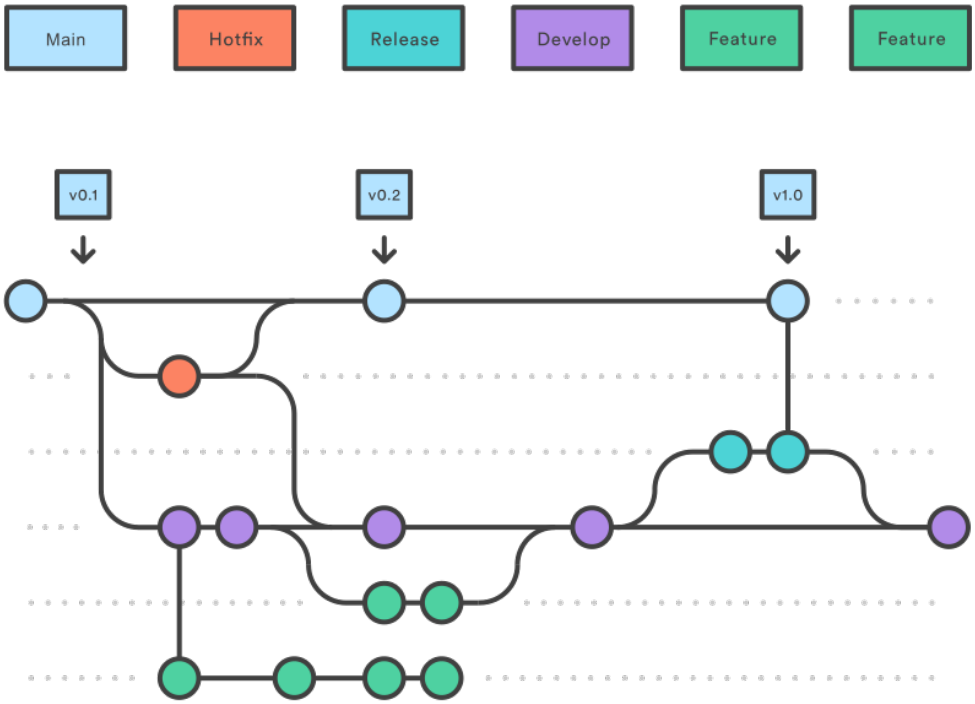

Figure 1.2: GitHub's Flow

1.2.1.4 How GitHub Supports Pure Storage's Workflow

An organization such as Pure Storage has several repositories for each project and its relevant code base. As each project has multiple scopes, a team would have partitions working on different features of a project which is supported by GitHub via branches that allow multiple unique versions to coexist alongside the main master branch. Teams may use GitHub's issues or an external issue tracking software such as Jira to keep track of tasks. The developers then add (commit) their code changes to their relevant branches; once a task (issue) is complete the branch containing all the updated additions (commits) can issue

a pull request. This pull request is assigned to a reviewer that reviews the code changes and tests it before merging it into the master branch

1.2.2 CI / CD

A CI / CD pipeline enables automation of the software delivery process which reduces the risk of manual errors and induces a more rapid development environment

1.2.2.1 Continuous Integration (CI)

CI is the process where developers merge numerous versions of their code regularly to a central repository; automated "unit" tests are then conducted on the code and builds are performed to the merged code on the master branch

1.2.2.2 Continuous Deployment (CD)

CD is the process of combing the code with the infrastructure ensuring it has passed further "integration" testing and policy checks after which it can be deployed (released) to the production environment for the end users

1.3 Architectural Patterns in Web Development

There are many architectural patterns that lie within the infrastructure of a web application in terms of its communication model between the client and the server as well as within the design of the web application itself

1.3.1 Client - Server

The Client - Server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers and the service requesters called clients [7]. Clients and servers exchange messages in a request-response cycle, where the client sends a certain request and the server receives that request, processes it and returns a response as seen in Figure 1.3

1.3.2 Model View Controller (MVC)

The MVC is an architectural design pattern that divides an application into 3 main interconnected elements; the model, the view and the controller where each component is built to handle a specific aspect of the application. The model is the central component of this architecture as it corresponds to all the data and its related logic, the view component is responsible for the presentation of the data to the user and the controller component is the interface between the model and the view components as it processes the incoming user

requests and handles the business logic of an application. This design pattern was originally intended for the use in desktop applications but has grown to become a widely adopted design pattern for web applications due to its easy maintenance, test-driven development approach as well as its high scalability

1.3.3 Component Based Architecture (CBA)

CBA focuses on the deconstruction of an application into individual functional or logical components. The primary objective of this design pattern is to ensure component re-usability, it also aligns the architectural view of a software system with the actual code view making it easier to understand and scale. A lot of web development frameworks support this design pattern as it provides a nicely layered architecture with each component encapsulated while providing flexibility as components don't have dependencies on each other

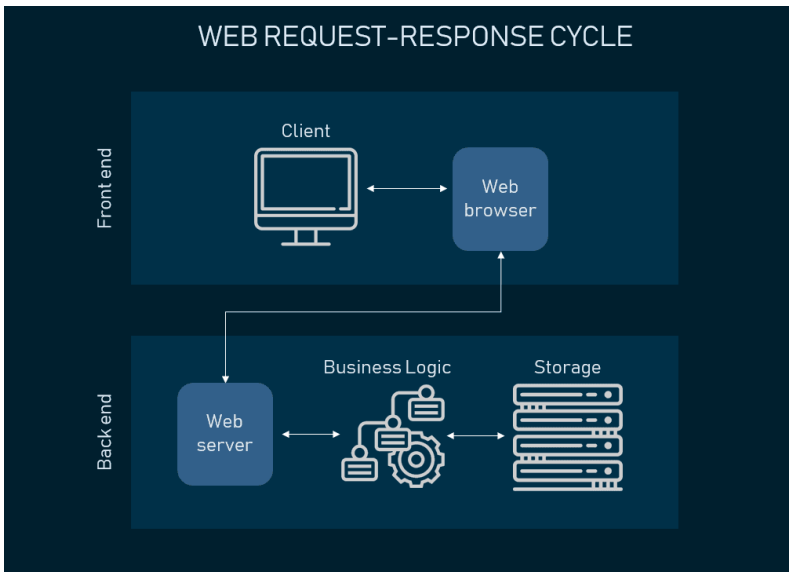

Figure 1.3: Request - Response Cycle

1.3.4 Single vs Multi Paged Applications

The main purpose of SPAs is the ability to access all the information from a single HTTP page; this implies that the application logic all lies within the client side using client side scripting technologies such as JavaScript and the server side is only being used for data storage. On the other hand, in MPAs request rendering happens every time a new page is requested from the server; meaning all of the processing is performed on the server's end, it receives a request, processes it according to the business logic and returns the relevant content to be displayed to the client. Commonly used server scripting

languages are PHP, Python and Java. The most notable difference between SPAs and MPAs is speed, SPAs have the advantage of requests being handled much faster mainly due to caching, they also have a versatile backend which can be reused for various mediums such as mobile apps. However, SPAs run on JavaScript which makes them prone to vulnerabilities such as Cross Site Scripting (XSS)

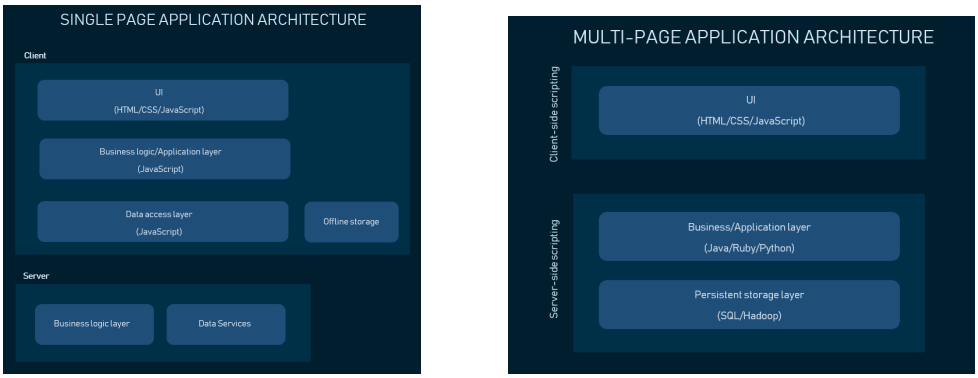

Figure 1.4: SPA vs MPA

1.4 Web App Development Frameworks

Web application frameworks are an essential part of any web development process as they provide a backbone that is designed to support web services, resources and APIs. Many web frameworks provide libraries for database access, templating, URL mapping, security and session management [8]. A commonly used architecture in web development frameworks is the MVC architecture which helps separate the data and business logic from the user interface which in turn modularizes the code in a structured manner and promotes code reuse

1.4.1 REST Architecture

Representation State Transfer is a software architectural pattern that defines a set of rules for creating web services, it provides certain standards that define how systems across the web should communicate with each other

1.4.1.1 Architectural Concept

The REST architecture is designed for network based applications such as client-server web apps. It emphasises a loose coupling between the client and server by creating a layer of abstraction and defining certain resources (entities) that can be accessed. Clients access these resources through requests

to URIs and the servers respond with a representation of the resource in hyper text format, which allows for the data to be interpreted by a wide number of client applications.

1.4.1.2 Architectural Constraints

The REST architectural pattern defines 6 main guiding constraints [9]:

- A "Uniform Interface" that implies there should be a uniform way of communicating with the server regardless of the device or application type
- "Statelessness" of the server as it retains no information about a client's previous requests for context. Each request made by the client must include all the information required for the server to fulfill the request
- "Cacheablility" of responses as each response should include whether it's cacheable or not as well as the duration of the caching on the client side
- A "Client - Server" architecture which enforces the principle of separation of concerns, where the client requests resources and is not concerned with the data storage and the server holds resources and is not concerned with the user interface or user state
- A "Layered System" which further separates the client and server as applications can be deployed on multiple servers each providing different resources where intermediary servers can enable load balancing
- "Code On Demand" which is an optional feature where servers can additionally provide executable code to the client such as Java applets or client side scripts such as JavaScript

1.4.1.3 Architectural Properties

When the REST architectural constraints are applied to a system it gains certain desirable non-functional requirements such as [10]:

- "Scalability" as the REST architecture supports load balancing via the layered system as well as the statelessness constraint which structures interactions as request-response pairs that are handled independently of other requests
- "Performance" as the REST architecture supports caching which helps keep the data more readily available and minimizes the overhead
- "Simplicity" due to the REST architecture's uniform interface where the functionality of services is abstracted to a simple interface

- "Modifiability" of the requirements as any architecture is bound to change over time, due to the loose coupling between components within this architecture changes can be incorporated with ease within it

1.4.2 Chosen Framework

I chose Django as my web app development framework for a number of reasons such as the fact that it follows the MVC architecture which encourages the creation of maintainable, reusable code as well as a test-driven development approach. It is also highly scalable as it supports a component based architecture which promotes the grouping of related functionality into reusable applications or components. Django also provides libraries for authentication, HTML templating, URL routing, session management and an Object Relational Mapper for interacting with the database

1.5 Web API

An Application Programming Interface is a set of constructs that allow developers to create complex functionality more easily by abstracting away the complexities of implementation and providing some syntax to be used in place [11]. A web API that follows the REST constraints is often referred to as a RESTful API, RESTful web APIs consist of one or more publicly exposed endpoints that define a request-response message system. These endpoints allow for access of resources by HTTP requests to certain URLs with encoded parameters after which JSON or XML is used to transfer the resource data

1.5.1 API Authentication

Since APIs have the ability to respond to protected resource requests they must have a way of authenticating whether the user trying to access the data is authentic, after which they can be authorized access to the protected resources. There are 3 main methods of authentication that are commonly used with APIs, basic HTTP authentication, API keys and OAUTH

1.5.2 OAUTH

OAUTH is a method used for authentication and authorization where a user requests authentication in the form of a token forwarded to the authentication server, after which the authorization server either rejects or accepts the request and issues an access token. This approach allows for even more control over the access of resources as it introduces scopes which is a list of permissions bound to an access token that are used to authorize access to different resources from the resource server

1.5.3 GitHub API

GitHub provides a RESTful API which has a list of endpoints giving access to the many resources available within GitHub such as repositories, issues, commits, pull requests and much more

1.5.4 GitHub Authentication

GitHub's API allows for 3 main ways of authentication, firstly the basic HTTP authentication method using a username and password which is not a good approach due to security vulnerabilities. The two other approaches rely on OAUTH to gain an access token and vary in the way it is obtained which is either through an OAUTH GitHub App where a user is redirected to a GitHub login page and grants the application access to the required scopes or through a GitHub App which is an installation made directly onto an organization or repository granting access to the defined scopes via the access token

1.5.5 Chosen Method of Authentication

Depending on the chosen method of authentication there could be two separate parts that require user authentication. One for accessing the web application itself, for example with a username and password and another for accessing GitHub's API. This is dependant on the way the authentication is approached as they could both be combined into a single "sign in using GitHub" redirect which authenticates users through an "OAUTH GitHub App" and its API keys. I have chosen to keep the authentications separate by firstly having a user authentication for accessing the web application via a username and password and then another for authenticating to the GitHub API via generating an access token with the necessary scopes to authorize and access the required endpoints needed to collect the suited data for this web app. The main reason for this is that from a design perspective this allows for scalability of the application later on via the integration of other APIs from the CI / CD pipeline

Chapter 2

Design

2.1 Technology Stack

During the development, I used PyCharm developed by JetBrains as it's a robust and extensive development environment for Python which includes many features such as code completion and debugging tools. As for the development using Python it was all done within a virtual environment to encapsulate all the dependencies of the project in a simple and accessible way. For the development of the web application; Django's web framework was used which provides a structured backbone for the implementation. The backend is written in Python using Django's built in libraries as well as some external Python libraries. The frontend is based of HTML5 pages that have CSS styling from the Bootstrap library and the database engine used throughout the development was an SQLite database

2.2 Django

This section provides an overall basis for creating a Django project and the applications (components) within that project as well as some CLI commands and their uses during the development for running and debugging the Django application. It also describes the workflow with the database as well as how to make use of the admin panel provided by Django

2.2.1 Useful CLI Commands

- To start off the development this bash command **django-admin startproject name** auto generates some code that establishes a Django application and its initial configuration such as the database configuration and some application specific settings. This provides the basis structured file tree for the root django application from which other applications (components) can be built and structured within

- Now that an initial project is setup you can start building apps (components) to encapsulate certain functionalities within the application by **python manage.py startapp name**

- To start a lightweight development web server on your local machine you can run the server by using **python manage.py runserver** which runs the application on the localhost via port 8000

- For debugging there are a couple useful commands such as running a python interpreter by **python manage.py shell** or a database command line client by **python manage.py dbshell**

- For inspecting the current database tables based off the models and migrations made you can run **python manage.py inspectdb**

2.2.2 Working with The Database

Django works with relational databases such as PostgreSQL or SQLite, it interfaces with them via Django's Object Relational Mapper (ORM) to modify, filter or delete instances of certain objects. During the development cycle a lot of changes are made to the models such as new fields being added or deleted or field types being changed. This is supported by Django through migrations that propagate the changes made to your models into the database schema. A common flow to follow for working with the database is after making any changes to the models to create migrations using **python manage.py makemigrations** and then to apply these migrations to the database schema using **python manage.py migrate**

2.2.3 Admin

Django comes automatically bundled with an admin dashboard panel that allows you to view and manage your model data in an intuitive way which is useful for debugging and testing the application. To gain access to the admin panel you need to create a superuser by running **python manage.py createsuperuser** which prompts you for a username and password which you can then use to login into the admin panel via **http://localhost:8000/admin**

2.3 Project Structure

After creating a project, an initial project structure is constructed after which other application (components) can be created within it as directories where each application has a similar file tree structure within

```
git ..... "GitHub" Application / Component
    migrations ..... Database Migrations
    admin.py ..... Admin Dashboard
    apps.py ..... Application Configuration
    forms.py ..... Forms
    models.py ..... Data Models
    tests.py ..... Tests
    urls.py ..... URL Routing
    views.py ..... Views
metrics ..... "Metrics" Application / Component
    migrations ..... Database Migrations
    admin.py ..... Admin Dashboard
    apps.py ..... Application Configuration
    forms.py ..... Forms
    models.py ..... Data Models
    tests.py ..... Tests
    urls.py ..... URL Routing
    views.py ..... Views
project ..... "Root" Django Application
    asgi.py ..... ASGI Server
    settings.py ..... Settings
    urls.py ..... URL Routing
    wsgi.py ..... WSGI Server
manage.py ..... Django's main script
```

2.3.1 Apps & Components

I chose to breakdown the architecture of my web application into two main components one that handles the user interface of the web application as well as the graphical data visualisations and another component for GitHub that handles all the data models and interacts with the API via the wrapper class to gather the data using certain specified logic. Within each application or component the MVC architecture is followed; this separation of the components promotes scalability of the application as integration with another API can be easily added by creating a new component to encapsulate that logic and then reusing the existing component for presenting the data through the user interface

2.4 Data Model

The data model consists of the data from the git application, the task worker data used for scheduling asynchronous tasks and the Dash application data. The user model has a username and password for logging into the application

as well as their access token for querying the GitHub API. Each user can then add as many repositories to their watchlist as they desire and each repository consists of many pull request waits that are used for the PR wait time metric as well as pull requests which are used for the PR size metric. The task worker model is used for scheduling periodic tasks as well as logging and monitoring their results. The Dash application model is used for storing the dynamic graphical application data

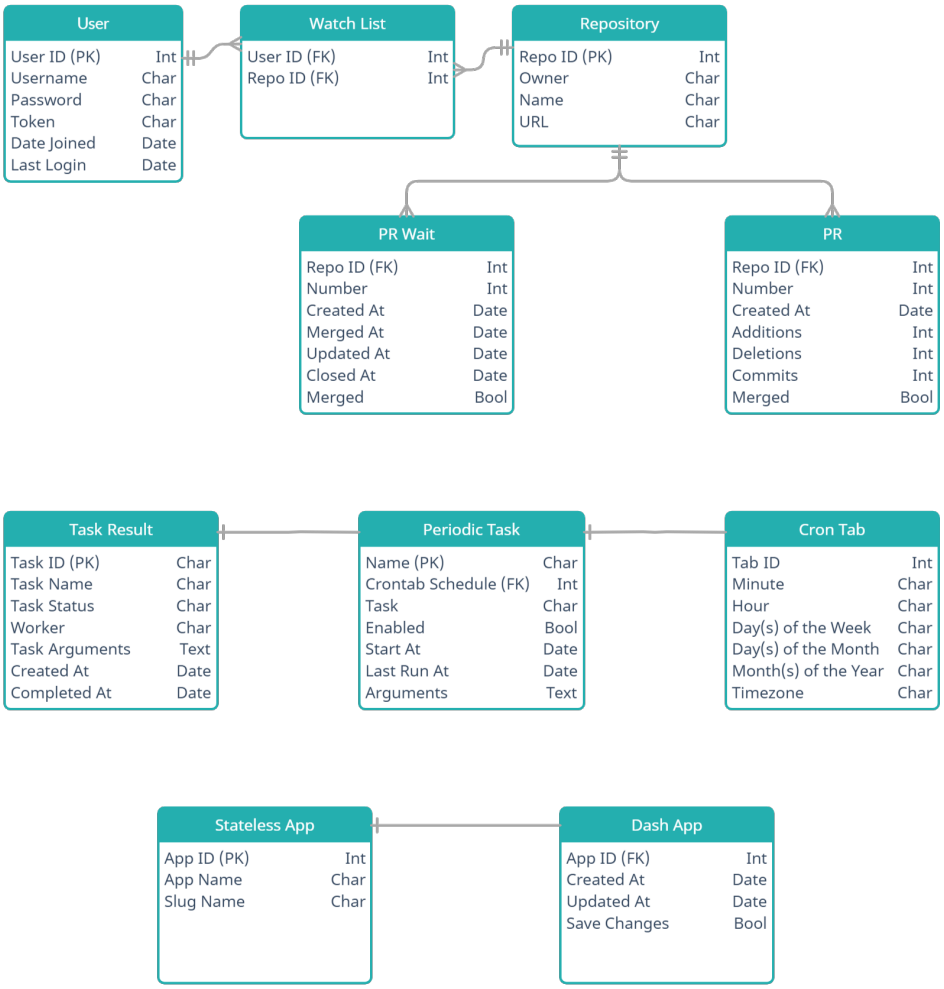

Figure 2.1: Data Model

2.5 User Interface

The UI was based of a bootstrap template that was adapted to fit the needs of the web application. There's a main landing page where users can login / register a new account. After which they are redirected to the homepage of the application where they are instructed on how to setup their GitHub access token. The sidebar is used to navigate to the different interfaces within the application, the profile page allows the users to quickly update and validate their GitHub access tokens, the main "dashboard" page provides a yearly overview or the PR wait time success rate, the "flow" page allows for a more detailed view of the metric as you can filter and view the success rate monthly, the "lifeline" page shows a weekly timeline of PR size metric. You can then edit your watchlist by removing existing repositories or adding new public GitHub repositories or any of your personal GitHub repositories from the "my repos" and "watchlist" pages

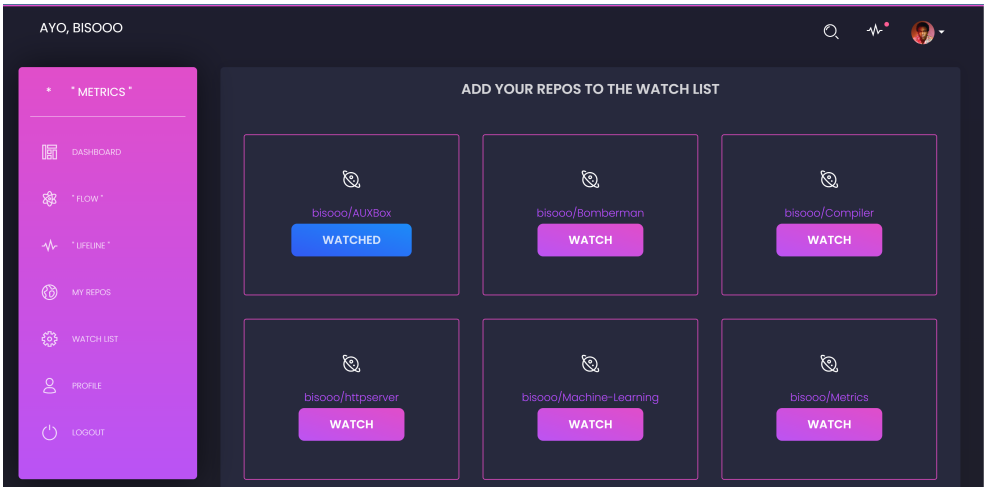

Figure 2.2: User Interface

2.5.1 Templates

Django uses templates as a convenient way to generate HTML dynamically, they provide a basis for the static content of a HTML page as well as syntax for inserting dynamic context into them. The most powerful aspect of Django's templating engine is the template inheritance which allows you to build a base "skeleton" template that includes all the common elements and static files of your user interface as well as defining certain "blocks" that child templates can then override. This basis of inheritance was used to build my user interface as the different pages of my web app all share a common theme and they

only differ in the main body content displayed within each of them. The dynamic content within the pages is passed to the templates via Django's views layer as a context dictionary of variables which can then be accessed, filtered and rendered using certain syntax within the HTML template. As seen in the template file in Listing 2.1 the inheritance from **base.html** simplifies the overall structure of the template where only certain blocks are overridden such as the main block content that encapsulates the main body of the HTML page, you can also see a use case for the context being passed from a view as the variable '**repos**' which is then iterated through using a for loop and filtered with an if statement

```
{% extends "base.html" %}
{% block title %} REPOS {% endblock %}
{% block content %}
<h4 class="title" style="text-align: center">
   ADD YOUR REPOS TO THE WATCH LIST
</h4>
<div class="card-body all-icons">
   <div class="row">
      {% for repo, watched in repos %}
      <div class="font-icon-list">
         <div class="font-icon-detail">
            <i class="tim-icons icon-planet"></i>
            <h5>
               <a target="_blank" href="{{ repo.html_url }}"> {{ repo.full_name }}</a>
               {% if watched %}
               <a class="btn btn-info"> WATCHED </a>
               {% else %}
               <a href="{% url 'repo_add' repo.owner repo.name %}" class="btn"> WATCH </a>
               {% endif %}
            </h5>
         </div>
      </div>
      {% endfor %}
   </div>
</div>
{% endblock content %}
```

Listing 2.1: Django - Templates

2.5.2 Static Files

Websites usually need to serve additional static files such as images, JavaScript or CSS. In Django this is managed by configuring a path to your static file directory which includes all the static files which can then be loaded into a template. This was used mainly in the base "skeleton" template where all the static file dependencies where loaded and are then inherited by the child template pages

2.5.3 Bootstrap

Django is mainly a more powerful backend web development framework, while it does provide certain libraries and functionalities that help with the frontend that is not its main domain. For that I used bootstrap which is one of the most popular frontend frameworks used by companies such as Twitter and Spotify. It consists of CSS classes and JavaScript code that can be integrated into the frontend by linking them within HTML elements

Chapter 3

Implementation

3.1 MVC Architecture

As previously mentioned Django follows an MVC design pattern, it uses slightly different terminology to name the different components where the model remains the same as Django's models, the view component is Django's templates and the controller is Django's views so essentially Django follows an Model-View-Template (MVT) design pattern

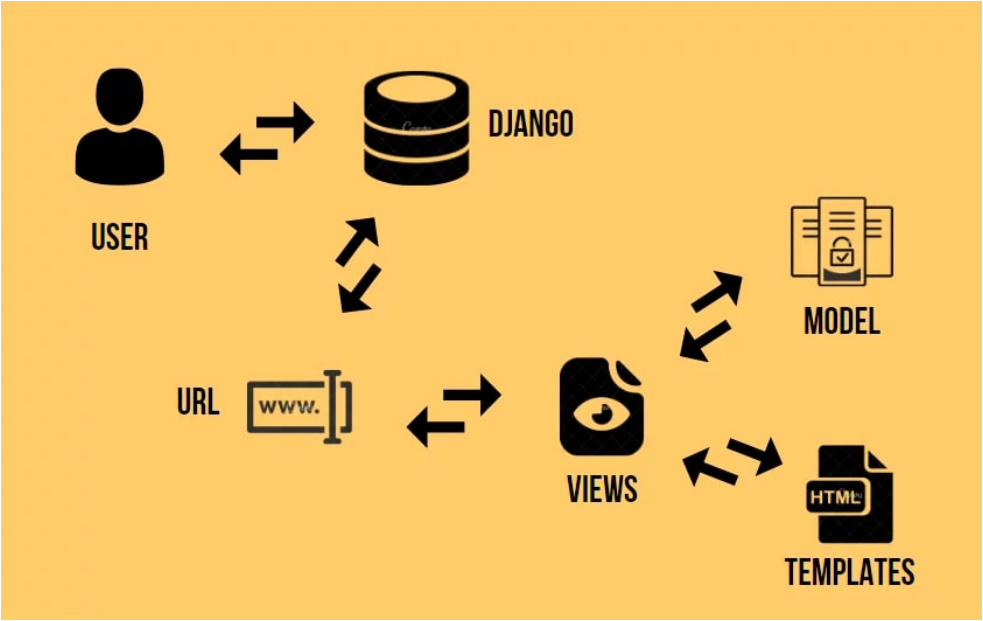

Figure 3.1: Django's MVC Architecture

3.1.1 Request Routing

A clean URL scheme is an important detail when building a web application, this is support by Django by mapping URL patterns to Django's views. When a user requests a page, Django sequentially checks it against all the URL patterns and stops at the first match, it then imports and calls the associated view and passes an instance of the HTTP request as well as any keyword arguments specified in the URL to it. The URL scheme was designed within my root Django application where the views from the two applications were imported and mapped to a suitable URL pattern as shown in Listing 3.1. Each mapping is also given a suitable name which can then be referenced within HTML elements in Django's templates for redirects as well as for passing keyword arguments as shown in Listing 3.2

```
from django.urls import path, include
# GIT
from git.views import register, login, logout
# METRICS
from metrics.views.homepage import homepage
from metrics.views.profile import profile
urlpatterns = [
# HOMEPAGE
path('', homepage, name='homepage'),
# PROFILE / LOGIN / REGISTER / LOGOUT
path('profile/', profile, name='profile'),
path('login/', login, name='login'),
path('register/', register, name='register'),
path('logout/', logout, name='logout'),
# WATCHLIST
path('watchlist/<int:repo_id>', delete_repo, name='delete_repo'),
...
]
```

Listing 3.1: Django - URLs

```
...
{% for i in watchlist %}
    <div class="font-icon-list">
        <div class="font-icon-detail">
            <i class="tim-icons icon-planet"></i>
            <h5>
                <a href="{{ i.repo.url }}"> {{ i.repo.owner }}/{{ i.repo.name }}</a>
            </h5>
            <a href="{% url 'delete_repo' i.repo.id %}" class="btn"> REMOVE </a>
        </div>
    </div>
{% endfor %}
...
```

Listing 3.2: Django - URL Mapping

3.1.2 Django's Views

Django's views is the "controller" component of the MVC architecture as it serves as a layer between the model component and the view component which is Django's templates. It receives the user's HTTP request handles it and returns a response. Besides the login and registration, most of the views lie within the metrics application as it's responsible for the user interface. Django provides two helper functions 'render' and 'redirect' that help span multiple levels of the MVC architecture. The 'render' function combines a given Django template with a context dictionary and returns an HTTP response as shown in Listing 3.3. The 'redirect' function as the name states returns an HTTP redirect response to the given URL passed as an argument. The main purpose of the view is to control the flow of the HTTP request-response cycle, the actual business logic is handled by another layer (services) where the views only contextualises the results from the business logic into context data passed to a Django template to be rendered in the HTTP response

```
from django.shortcuts import render, redirect
# GITHUB API LIBRARY SERVICES
from git.services.git import GitWrapper as git
def homepage(request):
    if not request.user.is_authenticated:
        return redirect('login')
    context = {}
    token = request.user.token
    valid_token = git(token).validate_login()
    if valid_token:
        context['valid_token'] = True
    return render(request, 'homepage.html', context)
```

Listing 3.3: Django - Views

3.1.3 Django's Forms

Any modern website supports user input that can be used to create, update or delete certain data model instances. This is supported by Django via forms. The two main areas that I used forms within my application were for the login and registration as they required user data that needs to be validated in order to create or update certain data model instances. Each Django form consists of the form fields as well as the data model object bound to those fields as shown in Listing 3.4. The form can then be rendered into a template while being protected by a CSRF token, once a user submits the form, the browser encodes the grouped form data and sends an HTTP POST request to the server which is then processed by a Django view, usually the same view that published the form it self as shown in Listing 3.5

```
# USER LOGIN FORM
class LoginForm(forms.Form):
    username = forms.CharField(max_length=20)
    password = forms.CharField(widget=forms.PasswordInput)
    model = User
    fields = ("username", "password")
    def clean(self):
        if self.is_valid():
            username = self.cleaned_data['username']
            password = self.cleaned_data['password']
            if not authenticate(username=username, password=password):
                raise forms.ValidationError("INVALID LOGIN")
```

Listing 3.4: Django - Forms

```
def login(request):
    context = {}
    if request.user.is_authenticated:
        return redirect('homepage')
    if request.POST:
        form = LoginForm(request.POST)
        if form.is_valid():
            username = request.POST['username']
            password = request.POST['password']
            user = authenticate(username=username, password=password)
            if user:
                auth_login(request, user)
                return redirect('homepage')
    else:
        form = LoginForm()
    context['login_form'] = form
    return render(request, 'login.html', context)
```

Listing 3.5: Django - Login View

3.1.4 Django's Models

Django models is the component used to design the data models for the database schema, it handles all the fields of a data model as well as their relations and characteristics. As shown in Listing 3.6 the one to many relationship between a repository instance and its pull request wait instances is achieved by having a foreign key field in the pull request wait model. Each field is assigned an appropriate type and certain fields have arguments such as blank or null that allow them to have specific characteristics

3.1.5 Django's Object-Relational-Mapper (ORM)

One of Django's most powerful features is its ORM which allows for interaction with an application data models for various relational databases such as PostgreSQL and SQLite. The ORM was used within the service layer where the

business logic of the application is developed. This abstraction gives you access to functions that let you create, filter, update and delete object instances as shown in Listing 3.7.

```
# PULL REQUEST WAIT MODEL
class PullRequestWait(models.Model):
    repo = models.ForeignKey(Repository, on_delete=models.CASCADE)
    number = models.PositiveIntegerField()
    created_at = models.DateTimeField()
    merged_at = models.DateTimeField(blank=True, null=True)
    updated_at = models.DateTimeField(blank=True, null=True)
    closed_at = models.DateTimeField(blank=True, null=True)
    merged = models.BooleanField()

    class Meta:
        unique_together = ["repo", "number"]
        verbose_name = "PR WAIT"
        verbose_name_plural = "PR WAITs"
```

Listing 3.6: Django - Model

```
def watchlist_remove(user, repo):
try:
    obj = WatchList.objects.all().get(user_id=user.id, repo_id=repo.id)
    obj.delete()
except WatchList.DoesNotExist:
    print("WATCHLIST OBJECT DOES NOT EXIST")

def watchlist_add(user, repo):
    try:
        obj = WatchList(user_id=user.id, repo_id=repo.id)
        obj.save()
    except (OperationalError, IntegrityError):
        print("DB ERROR")
        return HttpResponse('DB ERROR')

def repo_watched(user, owner, name):
    result = WatchList.objects.all().filter(user_id=user, repo__owner=owner, repo__name=name)
    if result:
        return True
    else:
        return False
```

Listing 3.7: Django - ORM

3.1.6 Services

The services layer was not in the initial Django structure of the application but was added during the development to encapsulate the business logic of the application, it consists of a list of functions that are grouped into packages.

There's a service package for the pull requests, repositories, users as well as one for general utility functions. The services layer also includes the Git wrapper class which is used for interacting with the GitHub API. The addition of this layer allowed for the abstraction of the business logic into these packages which can then be reused across the application

3.1.7 Django's Setting

The Django settings file lies within the root application and contains all the configuration options of the Django installation as well as some application specific settings. As new packages or libraries are added new module setting variables are edited or newly added such as the redis broker url, celery worker timezone, the custom user model to override the base user model as well as the task schedules for scheduling background tasks. Some module setting variables may contain sensitive information such as the secret key where a good practice is to create a **.env** file to store all your environment variables locally which can then be read into the settings file that way the sensitive information is not directly included in the source code

3.2 GitHub API Integration

After the basis of the application was structured, the main functionality came from integrating with the GitHub API. GitHub's API is a RESTful API that has certain endpoints exposed giving access to the many resources within GitHub, by going through the API's documentation I was able to authenticate users through the OAUTH authentication process and collect the data needed for the metrics from the API's endpoints into the database

3.2.1 GitHub API Library

Instead of building the wrapper class from scratch and having to directly construct the HTTP requests to the API as well as decode the JSON responses from the API, I used **github3.py** which is a wrapper for the GitHub API written in python as a basis to construct my own GitHub wrapper class. The wrapper class encapsulates a "GitHub" object which is the central point from where all the other functionality is accessed through its member functions

3.2.2 Authentication

Authenticating to the GitHub API was essential not only for allowing access to private user and organizational resources but also to avoid running into the API's rate limitations. The authentication was done using the access token that users had in their data model, when the wrapper class is initially constructed the login function provided by the GitHub API library is called which

authenticates the user and returns a GitHub object. The GitHub object is then stored as a member attribute in the wrapper class allowing class member functions to access the authorized GitHub object at any time to achieve their functionality

3.2.3 PR Wait Time

A member function of the wrapper class was developed to collect one year of pull requests for a repository into the database starting from the date/time at which the call to the function was made. Each pull request has certain attributes that were stored to be later used for devising the metric visualisations; The unique number that identifies the pull request on GitHub as well as the date and time it was created, updated, closed and merged at

3.2.4 PR Size

Another member function was developed to collect more detailed information for the last week of pull requests on a repository into the database starting from the date/time at which the call to the function was made. The attributes associated with this model were; The unique number that identifies the pull request on GitHub as well as the number of additions, deletions, commits and whether the pull request was merged or not

3.2.5 Task Queue

During the development phase, as I was testing interacting with the API on a big open source project I came across a major stumbling block which was that the interaction was all occurring within the request - response cycle of the web application which significantly decreased the responsiveness of the web application, this then lead to the introduction of a task queue that allowed for asynchronously executing time consuming tasks in the background outside of the request-response cycle which helped greatly increase the responsiveness of the web application. This was achieved by integrating with Celery which is a distributed task queue that focuses on real-time processing while also supporting task scheduling. A Redis message broker was hosted on Heroku that relayed the tasks to the Celery worker which then executed them. The task worker also allows for concurrency which enables running the tasks asynchronously using different “pools”. There are two main pool options one for multi-processing by creating child processes which is useful for CPU bound tasks and the one I used for this application which is for multi-threading and is useful for I/O bound tasks such as calling a web API. This multi-threading pool option makes use of so called “green threads” that are different than the conventional normal threads as their context switching is managed within the application space and not the operating system kernel space which further enhances their performance

3.2.6 Task Scheduler

There was a couple ideas initially about the way the "automation" of the data collection should be approached, one idea was to use webhooks that listen to certain actions such as a pull request being merged to notify the application to then directly query the API and update the relevant data. This approach was however neglected as webhooks are vulnerable to some security and network issues and therefore can't be mainly relied upon, they also would have to be directly setup onto a repository in order to function which poses as a kind of limitation to the set of repositories that can be used within the application. By incorporating the task worker the automated collection of data was enabled by making use of periodic tasks which can iteratively issue tasks every hour, day or sync to a timezone to trigger tasks at certain times, on certain days. I used this to update the pull request data in the database on a daily basis at midnight to keep the records in the database up to date with GitHub

3.3 Front End

After the integration with the GitHub API where the desired data was collected into the database it was time to to devise some suitable graphical visualisations to present the data in a useful way. This was achieved by integrating with a widely used Python graphing library called Plotly. The main Plotly library allows for the creation of static graphical visualisations but there's an extension called Plotly Dash that allows for creating interactive web graphical visualisations

3.3.1 Plotly Dash

The Django Plotly Dash library enables Plotly Dash applications to be served as part of a Django application. This works by wrapping around the Dash object and mapping the HTTP endpoints of the Dash object to the Django ones after which a Dash application can be embedded into a Django template through the use of template tags

3.3.2 Dash Apps

Dash Apps allow for live updating of an application's state by making use of Django's channels which extends the Asynchronous Server Gateway Interface (ASGI) to handle web sockets as well as a message broker backend such as Redis. This is achieved by using callback functions that are automatically called by Dash whenever an input component's value changes in order to update the output components. As seen in Listing 3.8 the callback function gets called whenever the input of selecting a target goal changes which then reevaluates the data and updates the graph

```
app = DjangoDash('yearly_pr_wait')
app.layout = html.Div([
    html.H5('YEARLY PR WAIT TIME'),
    html.P('SELECT A GOAL'),
    dcc.Slider(
        id='select-goal',
        marks={
            6: '6 HOURS', 12: '12 HOURS', 18: '18 HOURS',
            24: '24 HOURS', 30: '30 HOURS', 36: '36 HOURS',
            42: '42 HOURS', 48: '48 HOURS', 54: '54 HOURS',
            60: '60 HOURS', 66: '66 HOURS', 72: '72 HOURS'
        },
        min=1,
        max=78,
        value=72,
        step=None,
        updatemode='drag',
    ),
    dcc.Graph(id='slider-graph', animate=True),
])

@app.callback(
    Output('slider-graph', 'figure'),
    [Input('select-goal', 'value')])
def display_value(*args, **kwargs):
    ...
    fig = px.line(pr_wait, x="day", y=["success", "MA"])
    return fig
```

Listing 3.8: Plotly Dash - App

3.3.3 Metric Visualisation

As previously mentioned the way metrics are displayed is essential to capture the true essence of a metric in a useful way. During the development process, the data from the database was exported as a CSV file which was then imported into a Jupyter notebook to test out and analyse the different possible visualisations. Each data feature was analysed and then preprocessed to obtain the desired metric data, after which static Plotly visualisations were created. I then took the same logic and applied it within my application by building a Plotly Dash application that interfaced with the database and allowed for dynamic visualisations by incorporating user input. For the pull request wait time instead of presenting the actual time it took to complete the reviews, the user is allowed to set a goal of how long they would like to complete code reviews within, the review time data is then aggregated to show a success rate of how well the review times were relative to the goal as seen in Figure 3.2. A moving average is also used to smooth out the trend line and better contextualises how "successful" a data point is in relation to the data points before and after it as seen in Figure 3.3

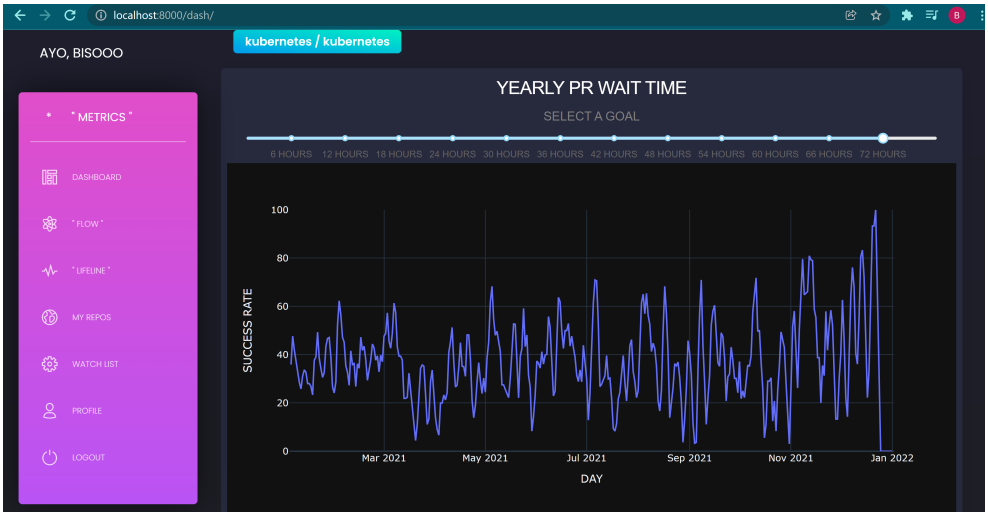

Figure 3.2: Yearly PR Wait Success Rate with a 72 Hour Goal

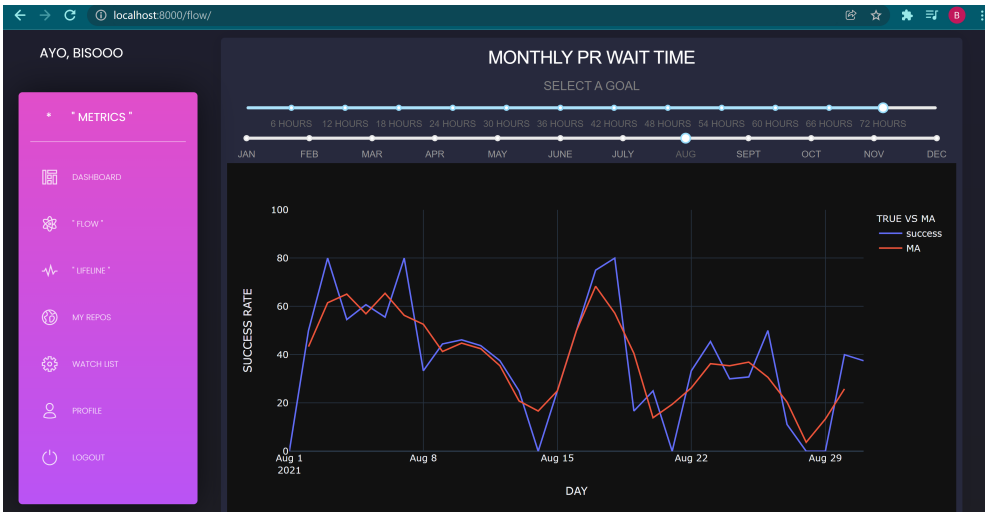

Figure 3.3: Monthly PR Wait Success Rate with a 72 Hour Goal

3.4 Security

Another essential bundled feature of the Django framework is that it comes equipped to handle many of the common security threats with websites such as SQL Injection (SQLi), Cross Site Scripting (XSS) and Cross Site Request Forgery (CSRF) [12]

3.4.1 SQL Injection (SQLi)

SQLi vulnerabilities enable malicious users to execute SQL code on a database allowing data to be accessed, modified or deleted irrespective of the user's permissions. As the main interaction with the database in Django is through models and the ORM, any raw SQL queries will be escaped by the database driver

3.4.2 Cross Site Scripting (XSS)

XSS is used to describe a class of attacks where an attacker injects client side scripts into the browsers of other users. This is usually achieved by storing malicious scripts into the database where they can be retrieved and displayed to other users. Django's template system protects against the majority of XSS attacks by escaping the specific characters that are dangerous in HTML

3.4.3 Cross Site Request Forgery (CSRF)

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent. Django provides protection against these attacks as any templates that use forms have a CSRF token tag which generates a user and browser specific key that is then validated with every HTTP POST request

Chapter 4

Testing

During the development I was using a test-driven approach of constantly testing the different units and functionality within my application as they were being developed. This was achieved through the python interactive shell or custom data through the admin panel. I also prepared a personal GitHub repository where I emulated actions to test capturing certain data through the GitHub API. Once I was happy with the initial functionality I would then test it on big open source GitHub repositories such as docker/compose and kubernetes/kubernetes. As I have collected data from a couple different open source repositories the one that deemed to be a more active project and had a large number of daily interactions was the kubernetes repository so I decided to test my application on it

Initially when a user first register they are advised on how to setup their GitHub access token and can then validate their token via the profile page which authorizes and authenticates that the token has the necessary scopes to access the API as show in Figure C.2 and Figure C.3

After the initial setup, the main flow of the web application is to add a GitHub repository via the watchlist page (Figure C.4) by its GitHub URL or repository name which are first validated (Figure C.5, Figure C.6) and then if valid another check occurs to see whether this repository already exists in the user's watchlist (Figure C.7) if not it's successfully added to their watchlist (Figure C.8, Figure C.9)

In the background this action triggers a check to see whether the data for this repository already exists in the database and if not a task is issued to the message broker hosted on Heroku which assigns the work to the Celery worker to collect and store the required data into the database asynchronously (Figure C.13, Figure C.16, Figure C.17). Once this is complete the data can then be graphically visualised in the user interface via the different dashboard

pages (Figure C.10, Figure C.11, FigureC.12)

Chapter 5

Future Work

During the analysis phase of this paper, realising the "cycle time" as the appropriate metric to measure productivity was essential. However, I was not able to capture the whole essence of the metric as it would require integration with many different tools and APIs so I decided to focus on a sub process within that which was the review time. To achieve a more complete measure of productivity, I think that integrating with the other tools in the CI / CD pipeline to obtain the two remaining sub processes within the "cycle time" which are the coding time and deploy time would help provide a more full resolved picture of the productivity of a development team. Integrating with those other CI / CD pipeline tools can also provide some other individual metrics which may be useful to measure and track such as the change failure rate or mean time to restore after a failure

Conclusion

The goal of this thesis was to discover a suitable way to measure productivity within the software development environment. By analysing the different methodologies used within the software development field and the many existing metrics which are currently used, understanding their benefits as well as their weaknesses, it prompted the question to reevaluate why productivity should be measured and what approach should be taken to capture productivity into a valuable metric. This resolved into the "cycle time" metric which essentially measures the process speed of how quickly a software development team can develop and deploy their ideas into production for their end users. The main metric that this paper focuses on is a sub process within the overall "cycle time" which was the code review time that was collected from GitHub, this part of the cycle falls within the continuous integration pipeline therefore visualising this metric would help teams better diagnose and resolve problems within their pipeline to speed up this process and in turn their overall "cycle time"

Since this web application was built from scratch I had the freedom to choose any technologies, by analysing different architectural patterns that are used in web development and their advantages I chose to use Django as my web development framework which allowed for a nicely structured MVC architecture that separates the data models and business logic from the user interface as well as a CBA that incentivized deconstructing the application into components based on their functionality. Having these two architectures in place while developing the application helped provide an easier development process as each newly added feature could be independently tested. It also allows for future scalability of the application as new components with additional functionality can easily be integrated into the currently existing architecture

The main bulk of the implementation was based around integrating with the GitHub API to collect and store the suited data into the database and then

building the suited graphical visualisations to display the metric data. The integration was done through a GitHub wrapper class where functions were designed to capture data from different endpoints of the API. During the development process, while testing some of the functionality that incorporated collecting data from the API for a big open source GitHub repository I came across a major hurdle which was that this process of interacting with the API was all occurring within the request-response cycle of the web application which served as a big problem for the responsiveness of the user interface. This then lead to the introduction of a message broker and a background worker that allowed for time consuming tasks to be run asynchronously in the background allowing for a much more responsive user interface. This also allowed for the automation of data collection from the API in the background to synchronize the data in the database on a daily basis

The building of suitable graphical visualisations was the final step within the development process of this web application. Each data feature was analysed to determine which would be useful for presenting the data, these were then preprocessed to capture the desired metric data. Then, by integrating with a widely used graphing tool I was able to create interactive dynamic visualisations that allowed for user input which influenced the resulting visualisations in real-time

There's a couple aspects of the application that I wish I had more time to better incorporate. One being a single unified dashboard page that allows for multiple visualisations to be structured out on a single page, as well as a way to allow users to customize different layouts. The other addition is for having an organizational login setup that allows organizations to register and setup their teams and projects accordingly where teams can then login and view their productivity or a project manager can login and track the productivity of the different teams across an organization

Bibliography

[1] Majewski, M. Top 6 software development methodologies - blog: Planview. Mar 2021. Available from: https://blog.planview.com/top-6-software-development-methodologies/

[2] Szalvay, V. Glossary of Scrum terms. Mar 2007. Available from: https://web.archive.org/web/20101129205330/http://scrumalliance.org/articles/39-glossary-of-scrum-terms

[3] Rubin, K. S. *Essential scrum: A practical guide to the most popular agile process*. Addison-Wesley, 2013.

[4] Pourshahid, A. Why cycle time may be the most important metric in software development. Feb 2017. Available from: https://www.klipfolio.com/blog/cycle-time-software-development

[5] About organizations. Available from: https://docs.github.com/en/organizations/collaborating-with-groups-in-organizations/about-organizations

[6] Introducing GitFlow. Available from: https://datasift.github.io/gitflow/IntroducingGitFlow.html

[7] Distributed Application Architecture. Available from: https://web.archive.org/web/20110406121920/http://java.sun.com/developer/Books/jdbc/ch07.pdf

[8] Web application framework. Available from: https://web.archive.org/web/20150723163302/http://docforge.com/wiki/Web_application_framework

[9] Rubin, K. S. *Essential scrum: A practical guide to the most popular agile process*. Addison-Wesley, 2013.

[10] Fadatare, R. Rest API - rest architectural properties. Jan 2021. Available from: https://www.javaguides.net/2018/06/rest-architectural-properties.html

[11] Introduction to web APIs. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction

[12] Security in Django. Available from: https://docs.djangoproject.com/en/4.0/topics/security/

Appendix A

Acronyms

API Application Programming Interface

CI Continuous Integration

CM Continuous Monitoring

CD Continuous Deployment / Delivery

CLI Command Line Interface

CBA Component Based Architecture

CSS Cascading Style Sheets

CSV Comma Separated Values

CSRF Cross Site Request Forgery

DevOps Development & Operations

HTTP Hyper Text Transfer Protocol

JSON JavaScript Object Notation

MVC Model View Controller

MVT Model View Template

MPA Multi Page Application

ORM Object Relational Mapping

OAUTH Open Authorization

PR Pull Request

REST Representational State Transfer

SQLi SQL Injection

SPA Single Page Application

URL Uniform Resource Locator

URI Uniform Resource Identifier

VCS Version Control System

XSS Cross Site Scripting

XML Extensible Markup Language

Appendix B

Installation Guide

- Using an IDE is highly recommended during the development as it provides a nice structured file tree overview and navigation. It also comes with tools such as code completion and debugging. I used PyCharm but any Python IDE should serve well
- Ensure you have an up to date version of Python installed on your machine, you can check this by running **python --version**, Python version 3.7 or higher is required
- Clone the repository code from **https://github.com/bisooo/Metrics** or if you already have it on your local machine then navigate into the root directory of the project
- You can then setup a Python virtual environment to install all the project dependencies into using **python -m venv name** where the 'name' specifies the path where you would like to setup your virtual environment directory
- To activate the virtual environment you can run **source name/bin/activate**
- After activating the virtual environment you can now install all the requirements into it using **pip install -r requirements.txt**
- To setup your Django environment there is a required secret key that you need to generate after which you can update the **.env** file to include that
- The last setup step that is required is migrating the models into the database schema which can be done by **python manage.py makemigrations** and then **python manage.py migrate**
- Finally, you can now run the server on your localhost at port 8000 using **python manage.py runserver**

- To setup the Redis message broker hosted on Heroku, head over to **https://dashboard.heroku.com/apps**, sign up and create a new app, choose a name for the app and the region where the server is hosted. After creating the app, head over to the resources tab and under Add-ons search for Heroku Redis and add that. Once it's succesfully installed open the add-on service and head over to the settings where you can view the datastore credentials. Copy the URI of the redis server and update the **.env** file to include that
- The celery worker will then receive the tasks from the message broker and can be hosted on your local machine to run asynchronous tasks by running **celery -A thesis worker -l info**
- The celery task scheduler can be run alongside that by running **celery -A thesis beat -l info**

Appendix C

Screenshots of the Web App

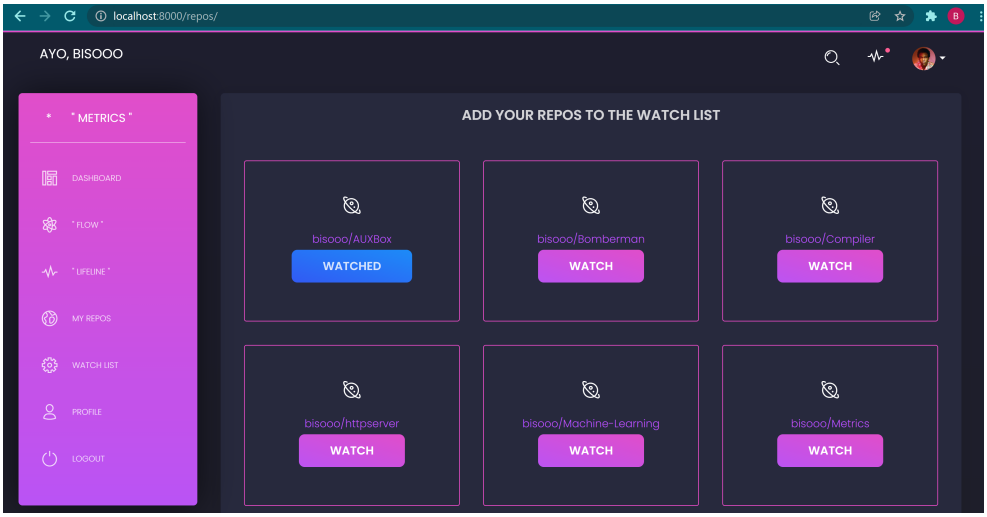

Figure C.1: My Repos - Watchlist

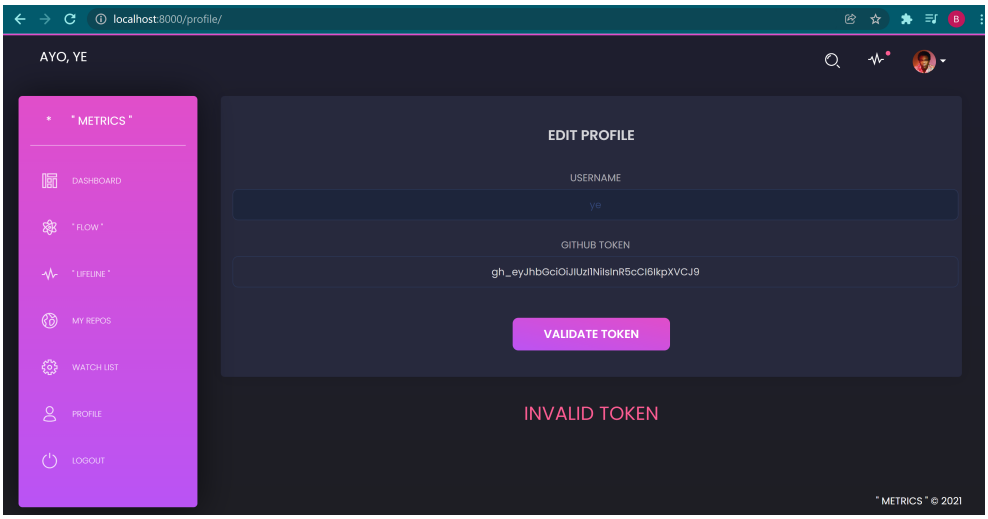

Figure C.2: Profile - Invalid Token

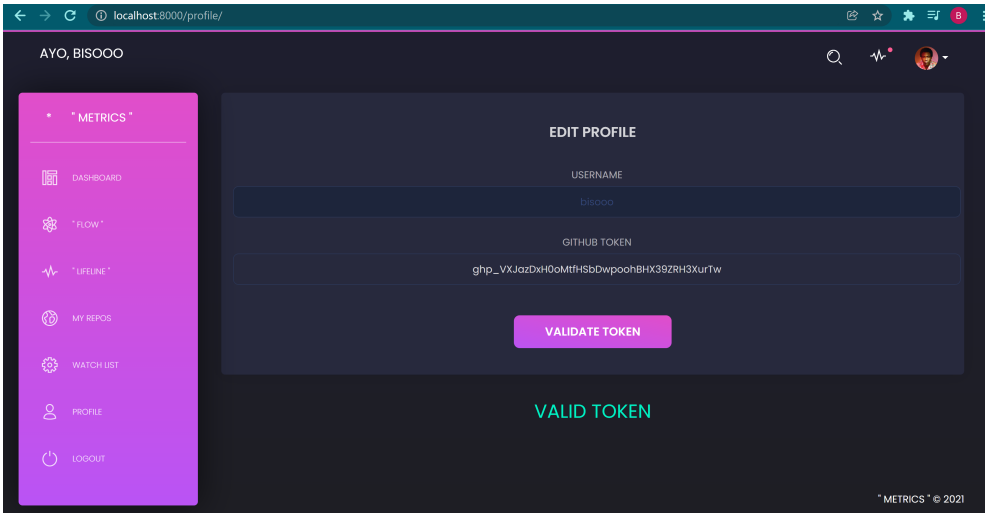

Figure C.3: Profile - Valid Token

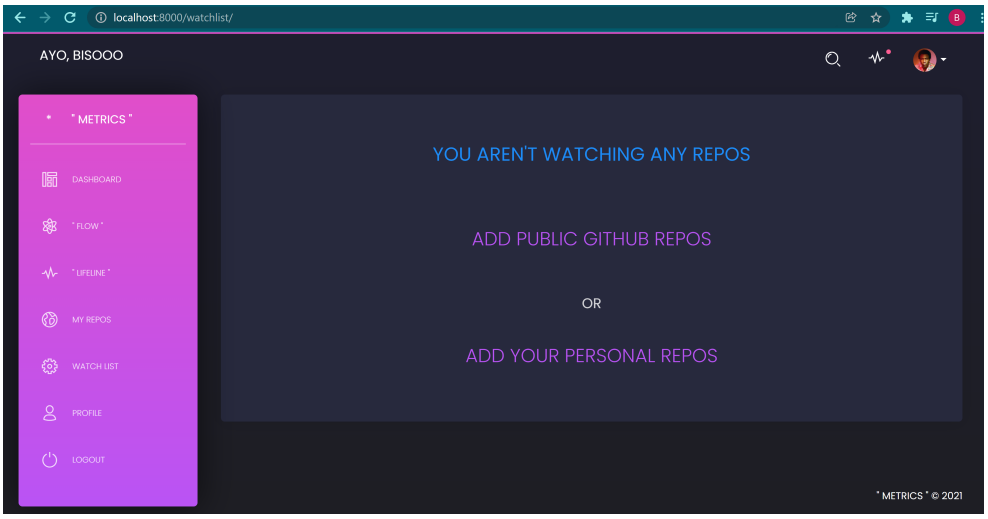

Figure C.4: Watchlist - Add a Repository

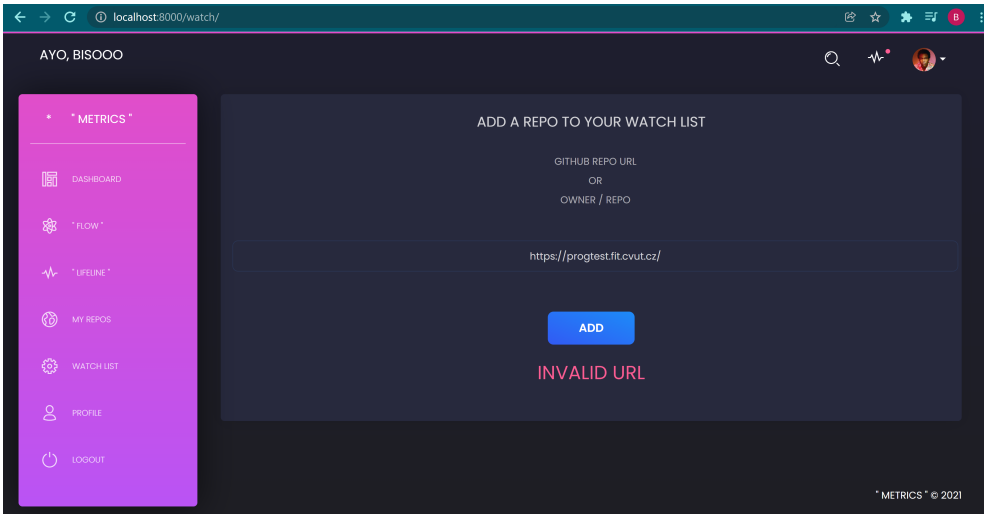

Figure C.5: Watchlist - Invalid Repository URL

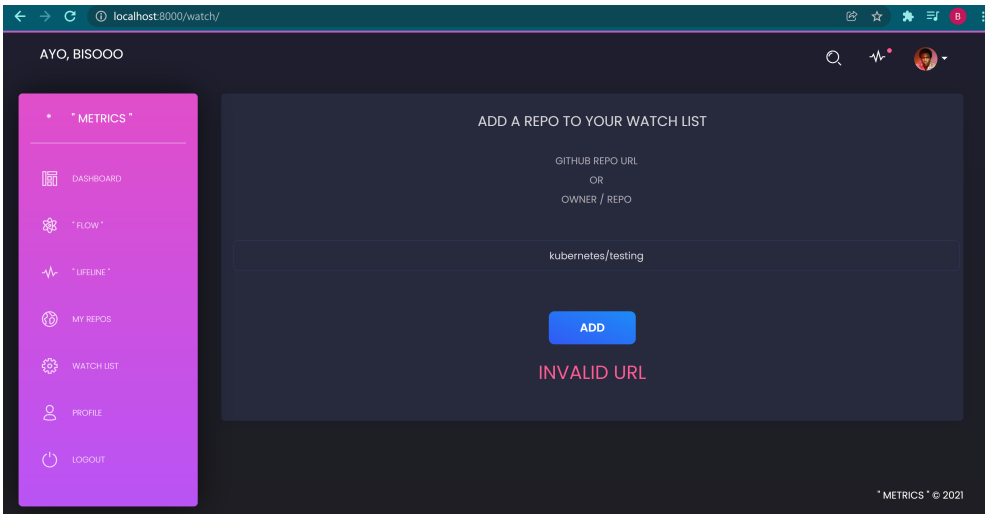

Figure C.6: Watchlist - Invalid Repository Name

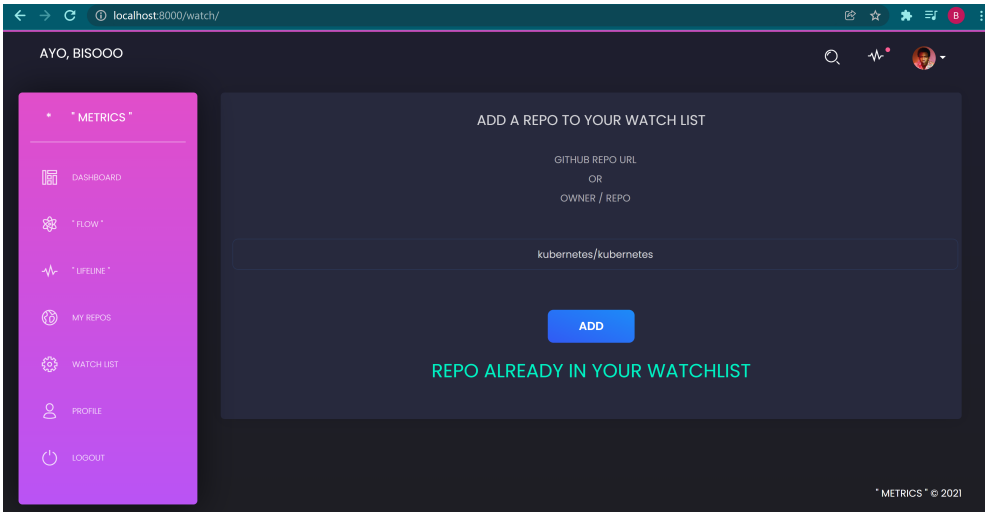

Figure C.7: Watchlist - Repository Already Exists

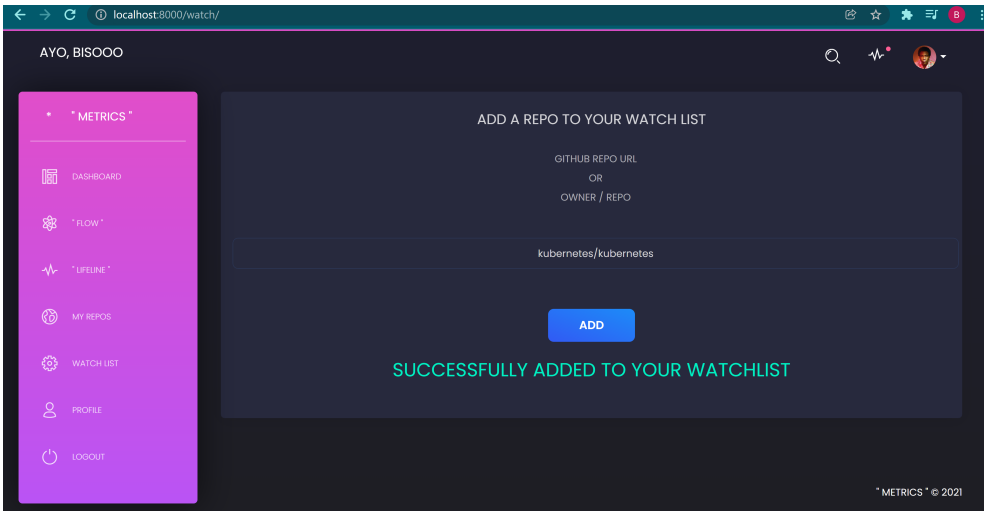

Figure C.8: Watchlist - Successfully Adding a Repository

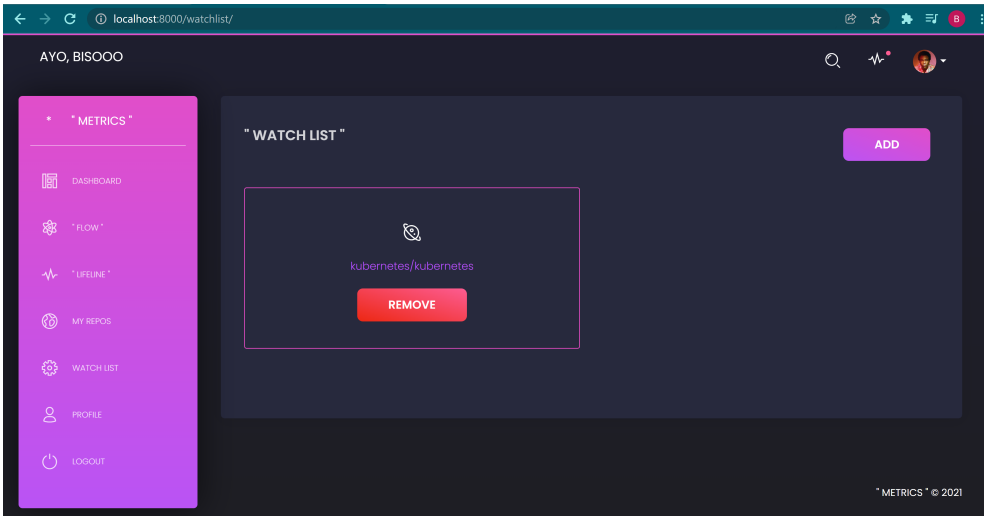

Figure C.9: Watchlist - Updated

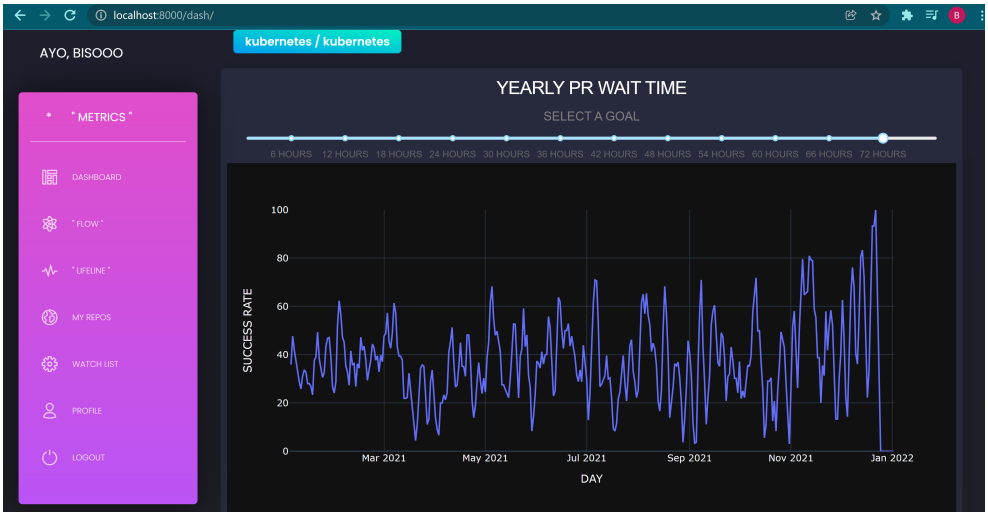

Figure C.10: Dashboard Visualisation

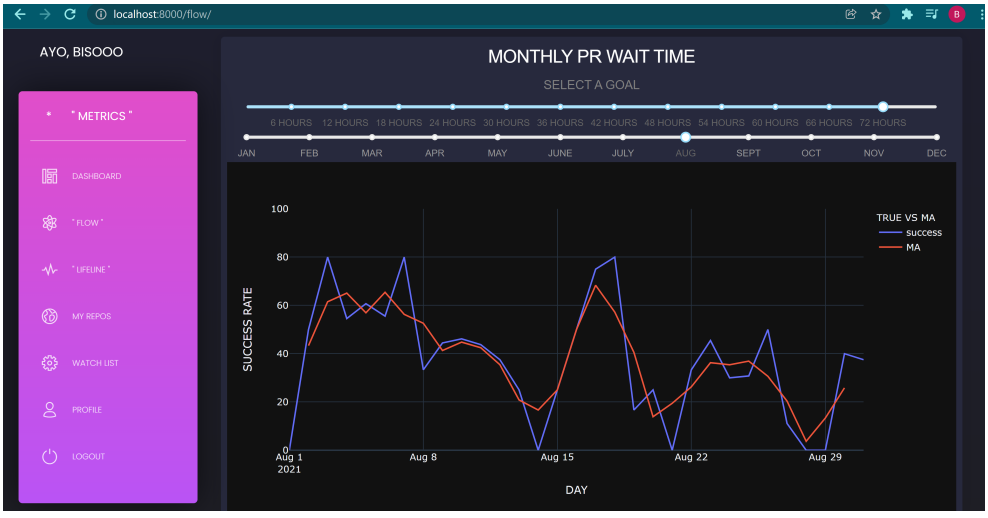

Figure C.11: Flow Visualisation

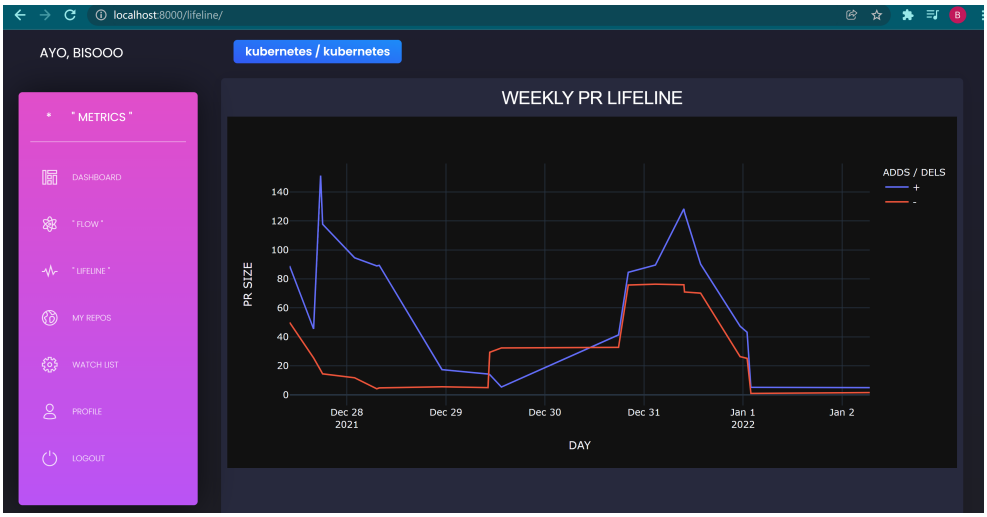

Figure C.12: Lifeline Visualisation

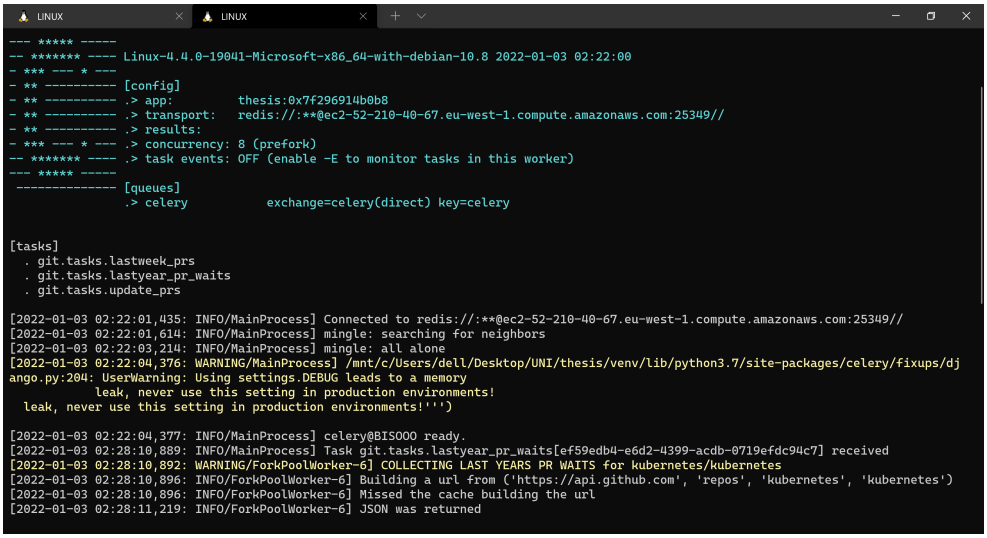

Figure C.13: Celery Worker - Collecting Yearly PR Waits

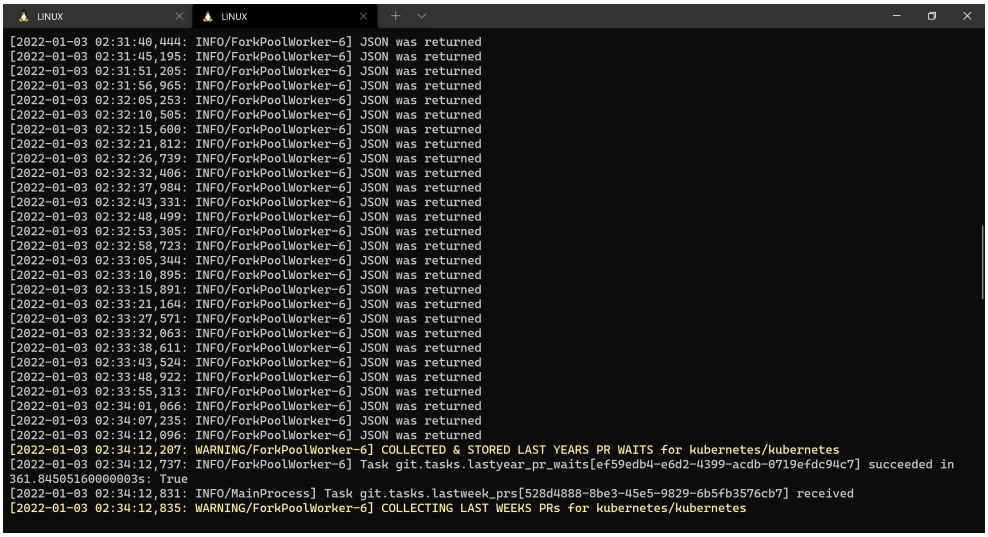

Figure C.14: Celery Worker - Yearly PR Waits Stored

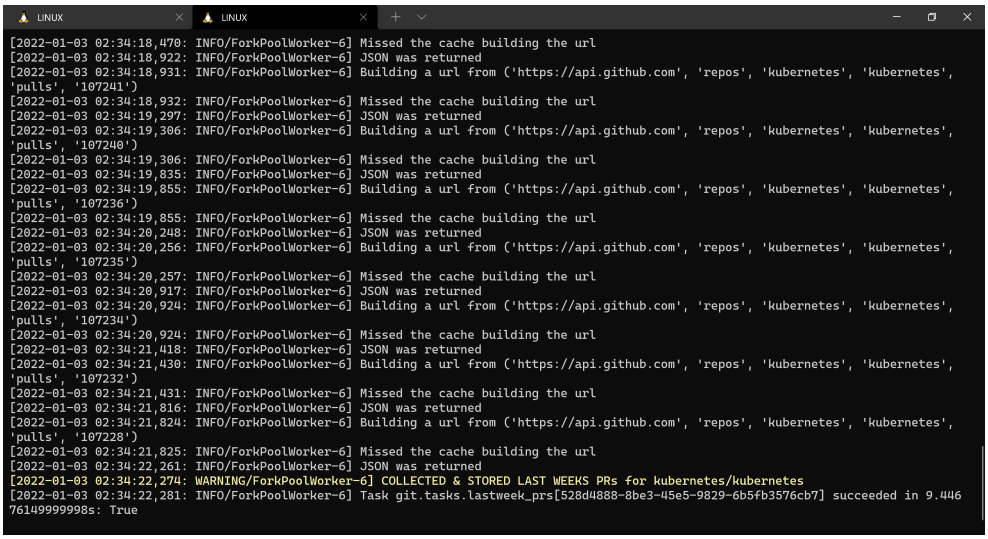

Figure C.15: Celery Worker - Weekly PRs Stored

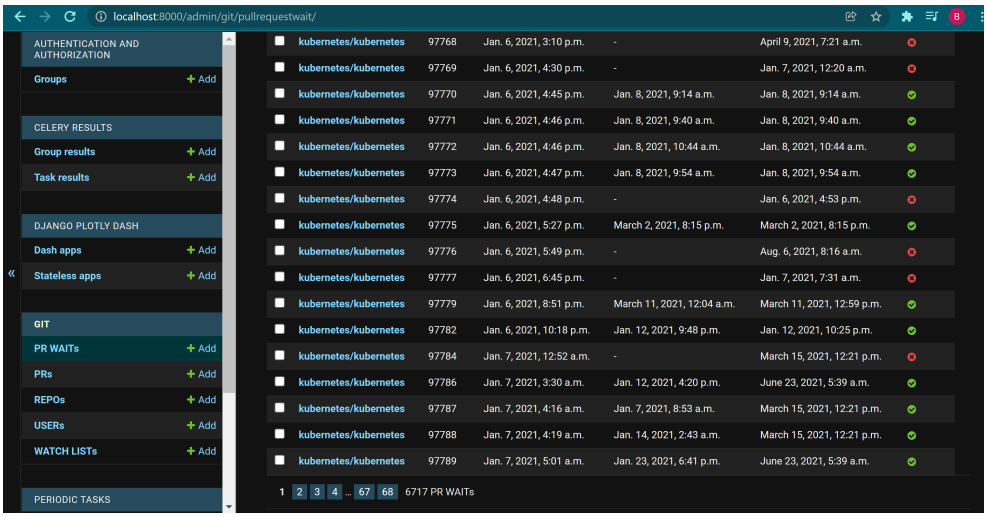

Figure C.16: Database - Yearly PR Waits

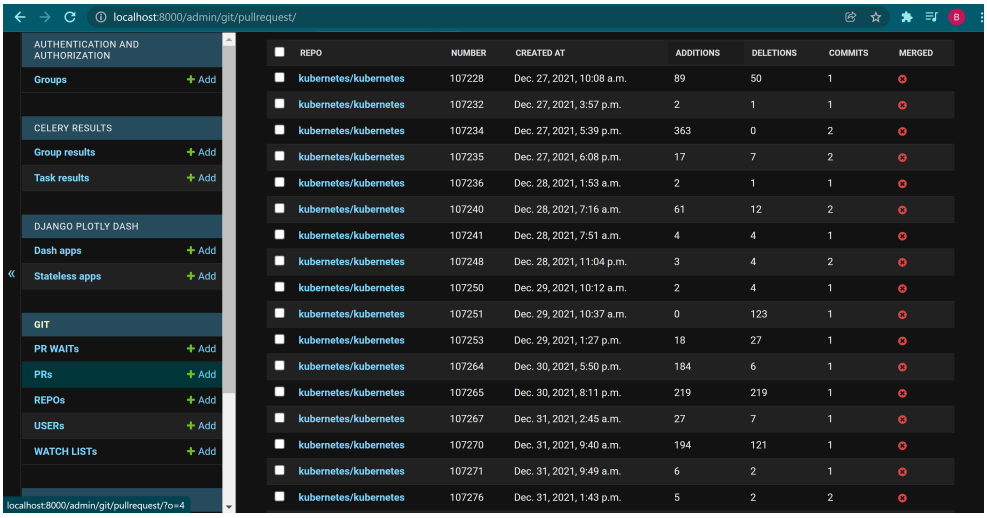

Figure C.17: Database - Weekly PRs

Appendix D

Contents of enclosed USB

- `thesis` — Web Application Source Code
  - `git` — "Git" Application / Component
    - `migrations` — Database Migrations
    - `services` — Service Packages
      - `git.py` — GitHub API Wrapper Class
      - `prs.py` — Pull Request Services
      - `repo.py` — Repository Services
      - `user.py` — User Services
      - `utils.py` — Utility Functions
    - `admin.py` — Admin Dashboard
    - `apps.py` — App Configuration
    - `forms.py` — Forms
    - `models.py` — Data Models
    - `tasks.py` — Asynchronous Worker Functions
    - `tests.py` — Tests
    - `views.py` — Views
  - `metrics` — "Metrics" Application / Component
    - `migrations` — Database Migrations
    - `static` — Static Files
    - `templates` — HTML Templates
    - `views` — Views
    - `visuals` — Plotly Dash Apps
    - `admin.py` — Admin Dashboard
    - `apps.py` — App Configuration
    - `models.py` — Models
    - `tests.py` — Tests
  - `thesis` — "Root" Django Application
    - `asgi.py` — Asynchronous Server Gateway Interface
    - `celery.py` — Asynchronous Worker Initialization
    - `routing.py` — Plotly Dash App Routing
    - `settings.py` — Settings
    - `urls.py` — URL Routing
    - `wsgi.py` — Web Server Gateway Interface
  - `.env` — Environment Variables
  - `manage.py` — Django's main script
  - `requirements.txt` — List of Python library dependencies
- `thesis.pdf` — Thesis text in PDF format